# 微服务

## 微服务（microservices）

### 微服务概述

微服务（microservices）是系统架构上的一种设计风格，它的主旨是将一个原本独立的系统拆分成多个小型服务，这些小型服务都在各自独立的进程中运行，服务之间通过基于HTTP的RESTful API进行通信协作。被拆分成的每一个小型服务都围绕着系统中的某一项或一些耦合度较高的业务功能进行构建，并且每个服务都维护着自身的数据存储、业务开发、自动化测试案例以及独立部署机制。由于有了轻量级的通信协作基础，所以这些微服务可以使用不同的语言来编写。

### 微服务与微服务架构

微服务，强调的是服务的大小，它关注的是某一个点，是具体解决某一个问题/提供落地对应服务的一个服务应用，狭意的看，可以看作Eclipse里面的一个个微服务工程/或者Module。

微服务架构是一种架构模式，它提倡将单一应用程序划分成一组小的服务，服务之间互相协调、互相配合，为用户提供最终价值。每个服务运行在其独立的进程中，服务与服务间采用轻量级的通信机制互相协作（通常是基于HTTP协议的RESTfulAPl)。每个服务都围绕着具体业务进行构建，并且能够被独立的部署到生产环境、类生产环境等。另外，应当尽量避免统一的、集中式的服务管理机制，对具体的一个服务而言，应根据业务上下文，选择合适的语言、工具对其进行构建。

### 微服务的优缺点

优点：

1、每个服务足够内聚，足够小，代码容易理解这样能聚焦一个指定的业务功能或业务需求

2、开发简单、开发效率提高，一个服务可能就是专一的只干一件事。

3、微服务能够被小团队单独开发，这个小团队是2到5人的开发人员组成。

4、微服务是松耦合的，是有功能意义的服务，无论是在开发阶段或部署阶段都是独立的。

5、微服务能使用不同的语言开发。

6、易于和第三方集成，微服务允许容易且灵活的方式集成自动部署，通过持续集成工具，如Jenkins,Hudson,bamboo。

7、微服务易于被一个开发人员理解，修改和维护，这样小团队能够更关注自己的工作成果。无需通过合作才能体现价值。

8、微服务允许你利用融合最新技术。

9、微服务只是业务逻辑的代码，不会和HTML，CSS或其他界面组件混合。、

缺点：

1、开发人员要处理分布式系统的复杂性

2、多服务运维难度，随着服务的增加，运维的压力也在增大

3、系统部署依赖

4、服务间通信成本

5、数据一致性

6、系统集成测试

7、性能监控……

## 微服务一些问题

### 什么是微服务？

### 微服务之间是如何独立通讯的

### springCloud和Dubbo有哪些区别？

1、传输协议

在spring cloud体系里面，所有RPC 都是基于HTTP协议的

Dubbo 里面，RPC 协议TCP 协议

HTTP 是建立在TCP 上的。(TCP 远远快于HTTP，大约7倍左右)

Dubbo 在java 体系，做远程调用很好

Spring Cloud 直接使用Http

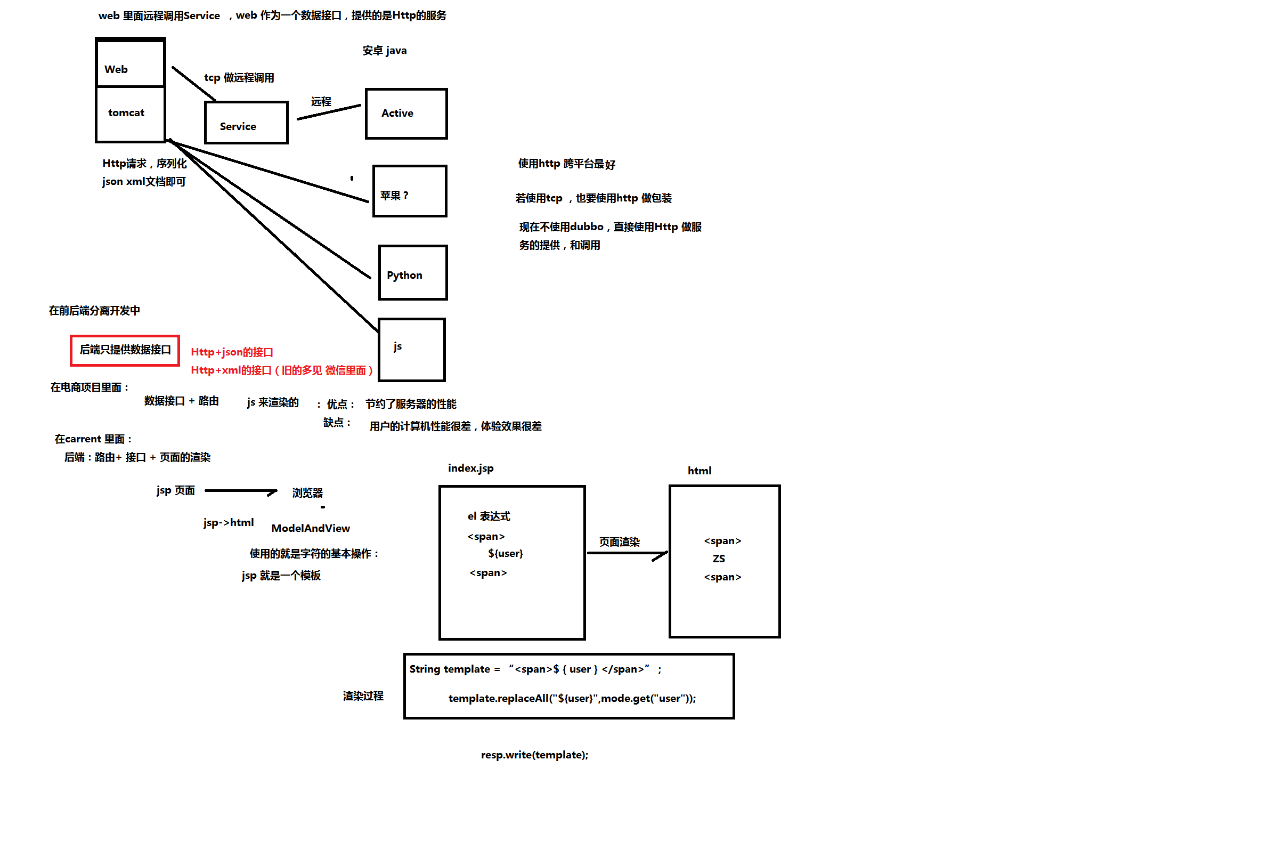
最大区别：SpringCloud抛弃了Dubbo的RPC通信，采用的是基于HTTP的REST方式。严格来说，这两种方式各有优劣。虽然从一定程度上来说，后者牺牲了服务调用的性能，但也避免了上面提到的原生RPC带来的问题。而且REST相比RPC更为灵活，服务提供方和调用方的依赖只依靠一纸契约，不存在代码级别的强依赖，这在强调快速演化的

微服务环境下，显得更加合适。

2、 http的跨平台

3、社区支持与更新力度

最为重要的是，DUBBO停止了5年左右的更新，虽然2017.7重启了。对于技术发展的新需求，需要由开发者自行拓展升级（比如当当网弄出了DubboX)，这对于很多想要采用微服务架构的中小软件组织，显然是不太合适的，中小公司没有这么强大的技术能力去修改Dubbo源码+周边的一整套解决方案，并不是每一个公司都有阿里的大牛+真实的线上生产环境测试过。





### SpringBoot和SpringCloud,请你谈谈对他们的理解）

1、SpringBoot专注于快速方便的开发单个个体微服务。

2、SpringCloud是关注全局的微服务协调整理治理框架，它将SpringBoot开发的一个个单体微服务整合并管理起来，为各个微服务之间提供，配置管理、服务发现、断路器、路由、微代理、事件总线、全局锁、决策竞选、分布式会话等等集成服

4、SpringBoot可以离开SpringCloud独立使用开发项目，但是SpringCloud离不开SpringBoot，属于依赖的关系。

5、SpringBgot专注于快速、方便的开发单个微服务个体，SpringCloud关注全局的服务治理框架。

### 什么是服务熔断？什么是服务降级

### 微服务的优缺点分别是什么？说下你在项目开发中碰到的坑

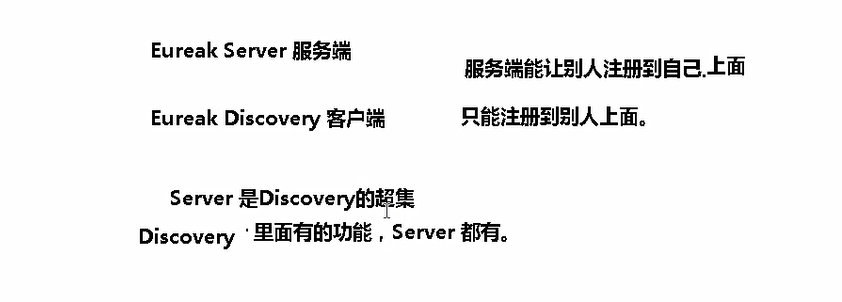
### 你所知道的微服务技术栈有哪些？请列举一二

### eureka和zookeeper都可以提供服务注册与发现的功能，请说说两个的区别？

# eureka

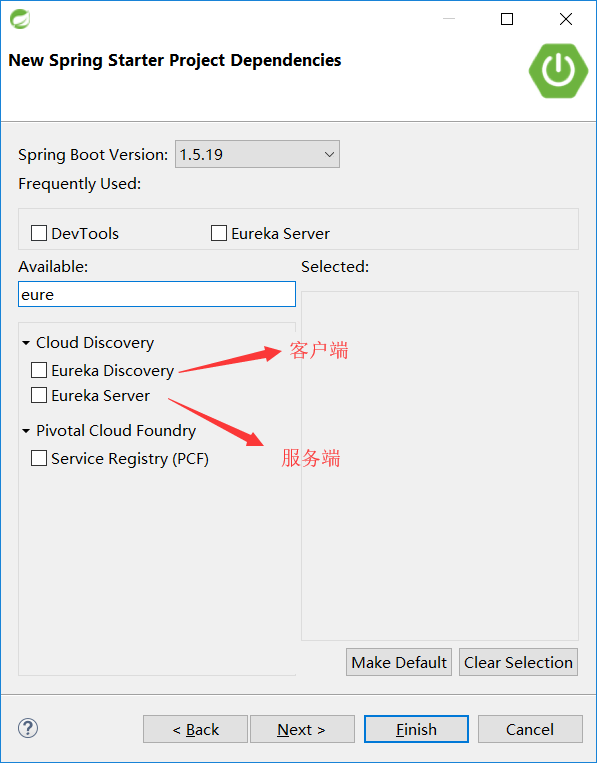
## eureka

### Eureak Server和Eureak Discover区别



## eureka的helloworld

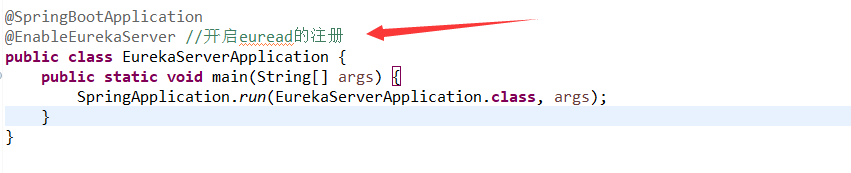
### 创建服务端项目



配置文件：yml

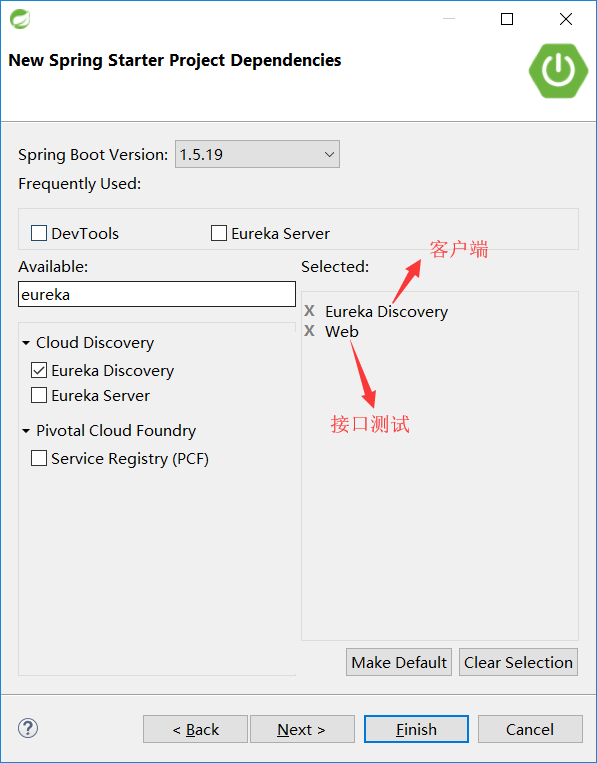
|  |
| --- |
| ## spring cloud eureka server 也有注册自己的功能，注册自己的功能默认开启  ## 自摸：自己注册自己  ## 指定该项目的运行端口  server:  port: 8761  #设置eureak的应用名字  spring:  application:  name: eureak-server  ## 设置注册到那个注册中心里面  eureka:  client:  ##eureak优化，关闭自己注册自己的功能,默认为true  register-with-eureka: **false**  ##eureak优化，是否抓取服务列表，因为此项目是注册中心，所以不需要去抓取  fetch-registry: **false**  service-url:    # 区域 ： 注册中心的地址  defaultZone: http://localhost:8761/eureka |

启动服务端：添加注解，开启eureka的注册服务



## eureka

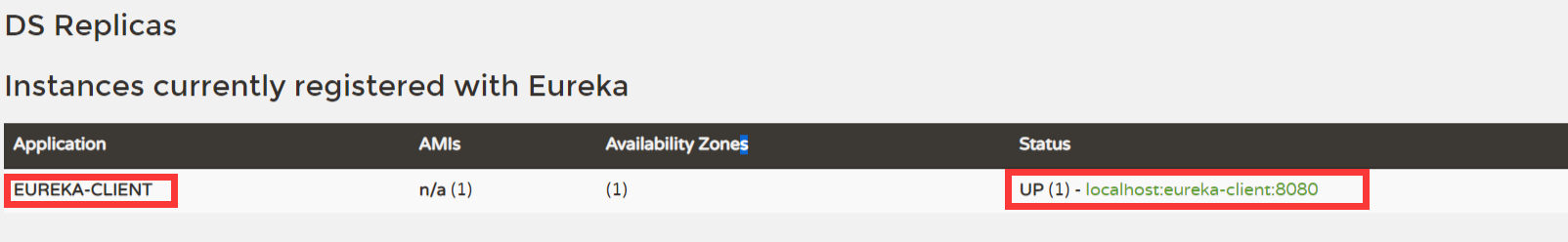
### 客户端



yml文件

|  |
| --- |
| ##配置eureka客户端的的地址  server:  port: 8080  ##配置eureak服务端注册中心的地址  eureka:  client:  service-url:  defaultZone: http://localhost:8761/eureka  fetch-registry: **true** ##从注册中心获取服务列表,默认为true  register-with-eureka: **true** ##注册自己到注册中心，默认为true  instance:  prefer-ip-address: **true** ##使用ip地址代替计算机的名称，点击status显示的地址  instance-id: eureka-client ##服务status所显示的内容  home-page-url: https://www.baidu.com/ ##并不是点击实例instance-id的连接，  ##而是在监控时，使用boot-admin做监控时显示的信息    ##告诉eureka我的服务名称  spring:  application:  name: eureka-client |

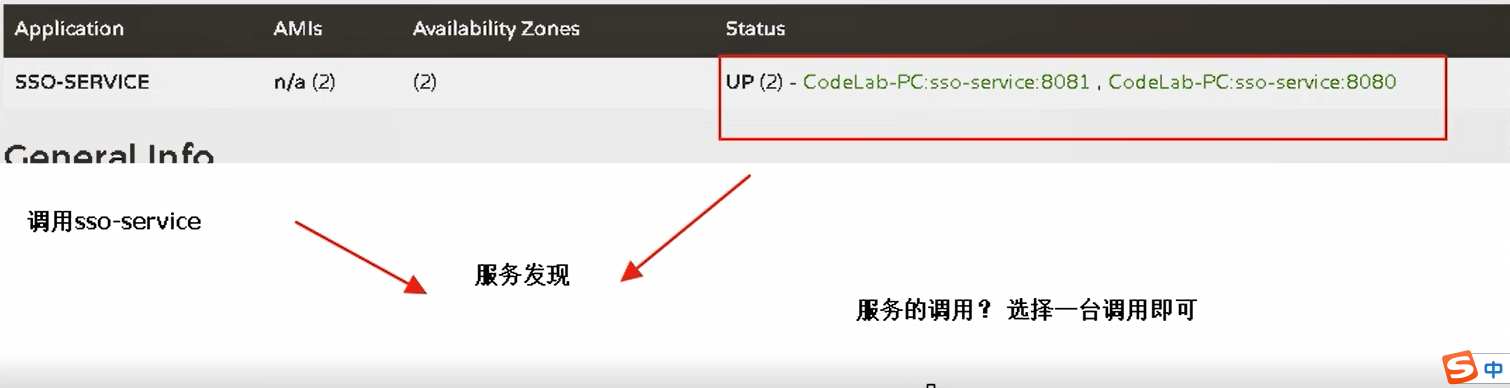
启动：服务上线



status：默认的访问

：<http://localhost:8080/info> 可以写个controlller 通过这个路径访问到服务端。

### 服务调用

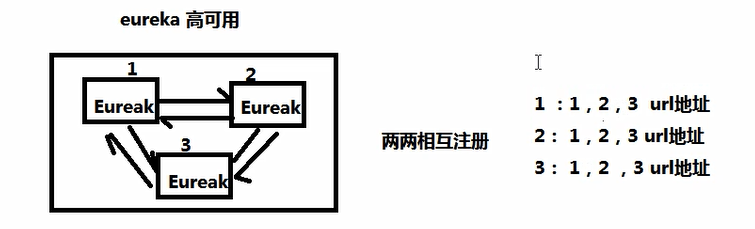


### 配置文件的优化

|  |
| --- |
| instance:   lease-renewal-interval-in-seconds: 2 # 设置心跳的时间间隔（默认是30秒）   lease-expiration-duration-in-seconds: 5 # 如果现在超过了5秒的间隔（默认是90秒）   instance-id: cloud-provider-client.com # 在信息列表时显示主机名称(设置)  prefer-ip-address: true # 访问的路径变为 IP 地址 （设置） |

## eureka集群

### 集群方案：



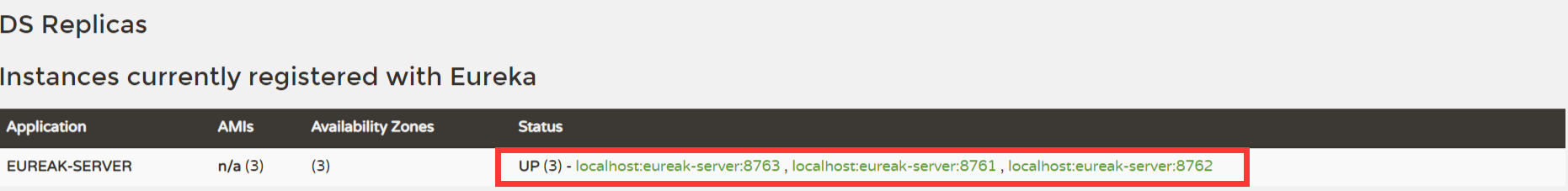
|  |  |  |
| --- | --- | --- |
| Ip | 端口 | 编号 |
| Localhost | 8761 | Eureka-1 |
| Localhost | 8762 | Eureka-2 |
| Localhost | 8763 | Eureka-2 |

### 配置文件

|  |
| --- |
| ## spring cloud eureka server 也有注册自己的功能，注册自己的功能默认开启  ## 自摸：自己注册自己  ## 指定该项目的运行端口  server: ##使用三元表达式，如果没有SERVER\_PORT则使用8761  ##该值以后来自环境变量，在docker启动时，通过-e参数指定  port: ${SERVER\_PORT:8761},${SERVER\_PORT:8762},${SERVER\_PORT:8763}    ##设置eureak的应用名字  spring:  application:  name: eureak-server  ##设置注册到那个注册中心里面  eureka:  client:  ##eureak优化，关闭自己注册自己的功能,默认为true,集群是要开启  register-with-eureka: **true**  ##eureak优化，是否抓取服务列表,集群时，要开启，要拿到其他注册中心的服务  fetch-registry: **true**  service-url:    ##区域 ： 注册中心的地址,该值以后来自环境变量，在docker启动时，通过-e参数指定  defaultZone: ${EUREKA\_SERVER\_URL:http://localhost:8761/eureka},${EUREKA\_SERVER\_URL:http://localhost:8762/eureka},${EUREKA\_SERVER\_URL:http://localhost:8763/eureka} |

### 启动

启动时分别使用不同端口启动，



# 客户端负载均衡Ribbon

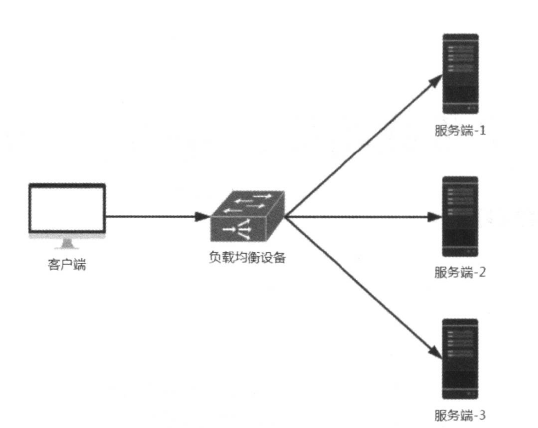
## ribbon负载均衡

### 负载均衡

负载均衡在系统架构中是一个非常重要，并且是不得不去实施的内容。因为负载均衡

是对系统的高可用、网络压力的缓解和处理能力扩容的重要手段之一。我们通常所说的负载均衡都指的是服务端负载均衡，其中分为硬件负载均衡和软件负载均衡。硬件负载均衡

主要通过在服务器节点之间安装专门用于负载均衡的设备，比如F5等；而软件负载均衡则是通过在服务器上安装一些具有均衡负载功能或模块的软件来完成请求分发工作，比Nginx等。不论采用硬件负载均衡还是软件负载均衡，只要是服务端负载均衡都能以类似下图的架构方式构建起来



硬件负载均衡的设备或是软件负载均衡的软件模块都会维护一个下挂可用的服务端清单，通过心跳检测来剔除故障的服务端节点以保证清单中都是可以正常访问的服务端节点。当客户端发送请求到负载均衡设备的时候，该设备按某种算法（比如线性轮询、按权重负载、按流量负载等）从维护的可用服务端清单中取出一台服务端的地址，然后进行转发。

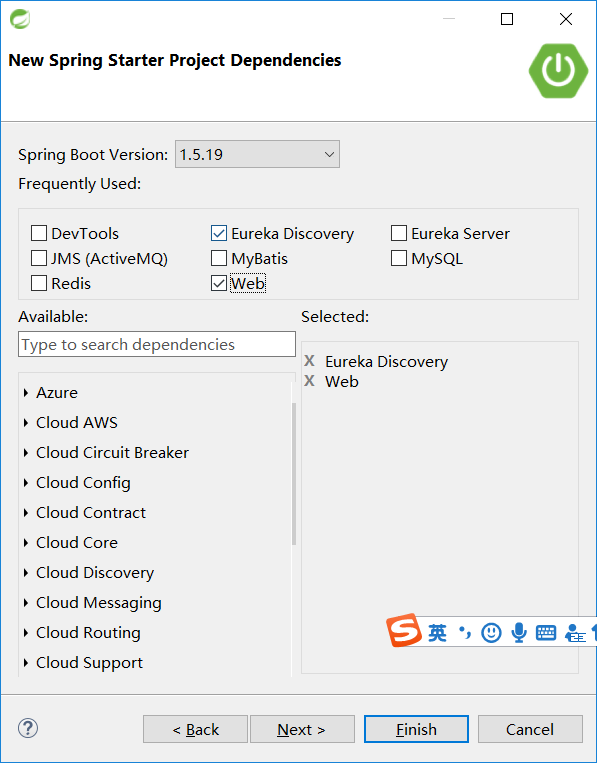
### 客户端负载均衡的使用

服务提供者只需要启动多个服务实例并注册到一个注册中心或是多个相关联的服务注册中心。服务消费者直接通过调用被@LoadBalanced注解修饰过的RestTemplate来实现面向服务的接口调用。这样，我们就可以将服务提供者的高可用以及服务消费者的负载均衡调用一起实现了。

## helloword

### teacher-service

创建项目



### controller

|  |
| --- |
| @RestController  **public** **class** TeacherController {  //远程调用userController  @Autowired  **private** RestTemplate restTemplate;  @GetMapping("/rpc")  **public** String rpc() {  String userJson = restTemplate.getForObject("http://localhost:8080/user/1", String.**class**);  //获取一个用户  **return** userJson+"用户的json";  }  } |

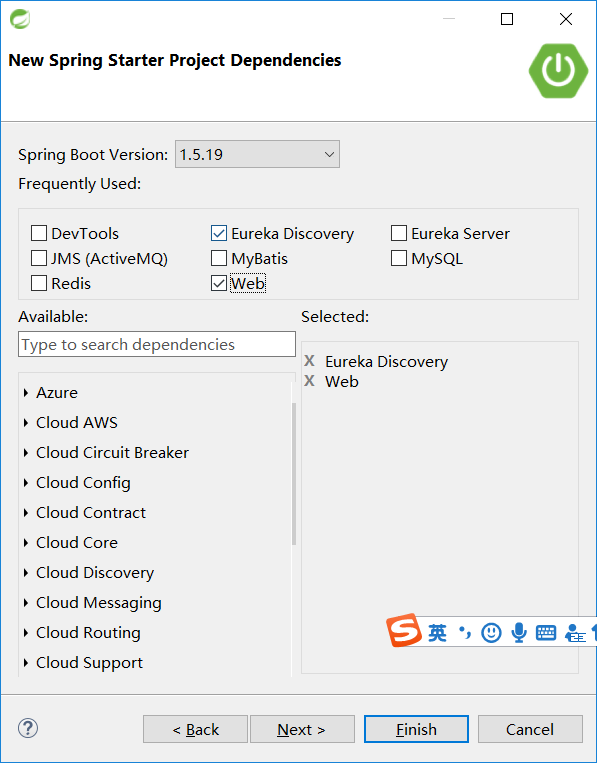
将<http://localhost:8080/user/1> 换为<http://user-serice/user/1>

主机名+端口换为服务的名字，来调用服务

teacher restTemplate

|  |
| --- |
| @LoadBalanced //使用该注解实现，服务的负载均衡  @Bean  **public** RestTemplate restTemplage() {  **return** **new** RestTemplate();  } |

### user-service1



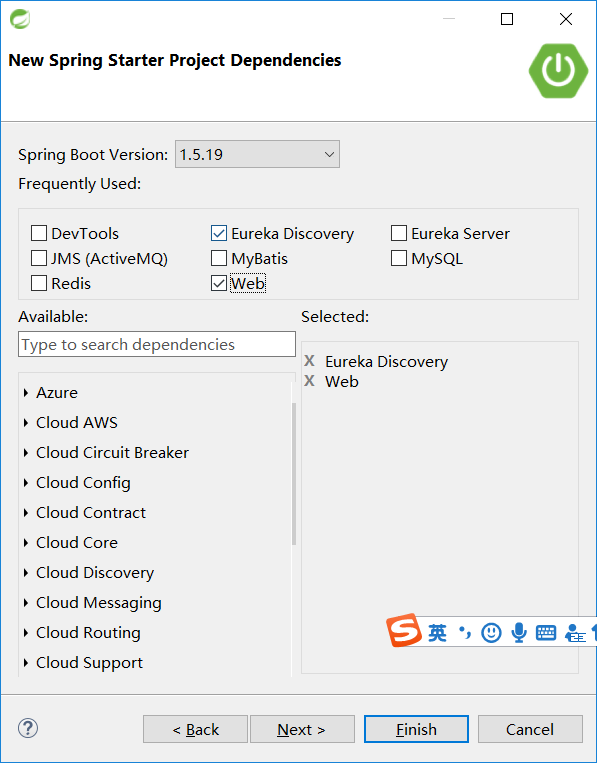
yml文件

|  |
| --- |
| server:  port: 8080    eureka:  client:  service-url:  defaultZone: http://localhost:8761/eureka/  spring:  application:  name: user-service |

controller

|  |
| --- |
| @RequestMapping("/user")  @RestController  **public** **class** UserController {  @GetMapping("/{id}")  **public** User getUser(@PathVariable("id")Integer id) {  System.***out***.println("user1");  System.***out***.println("被调用");  **return** **new** User("sx", "123456");  }  } |

### user-service2



yml

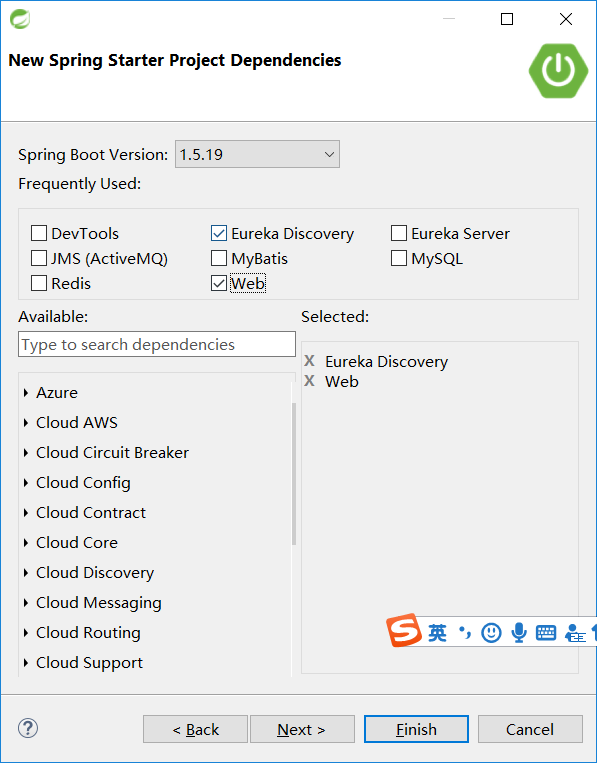
|  |
| --- |
| server:  port: 8100    eureka:  client:  service-url:  defaultZone: http://localhost:8761/eureka/  spring:  application:  name: user-service |

controller

|  |
| --- |
| @RequestMapping("/user")  @RestController  **public** **class** UserController {  @GetMapping("/{id}")  **public** User getUser(@PathVariable("id")Integer id) {  System.***out***.println("user2");  **return** **new** User("sx", "123456");  }  } |

## user-service

### 创建两个user-service项目

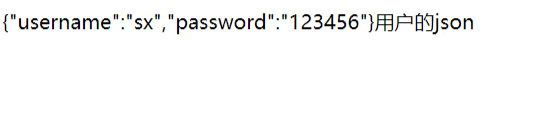


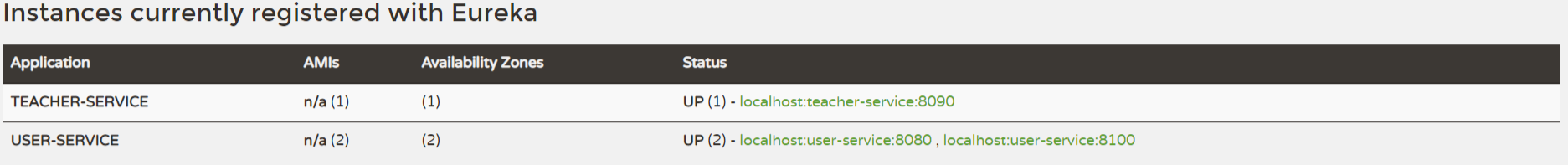
### controller

|  |
| --- |
| @RequestMapping("/user")  @RestController  **public** **class** UserController {  @GetMapping("/{id}")  **public** User getUser(@PathVariable("id")Integer id) {  System.***out***.println("被调用");  **return** **new** User("sx", "123456");  }  } |

### 访问<http://localhost:8090/rpc>

返回结果





### 结果

1、能够调用服务

2、相同的服务名之间实现负载均衡

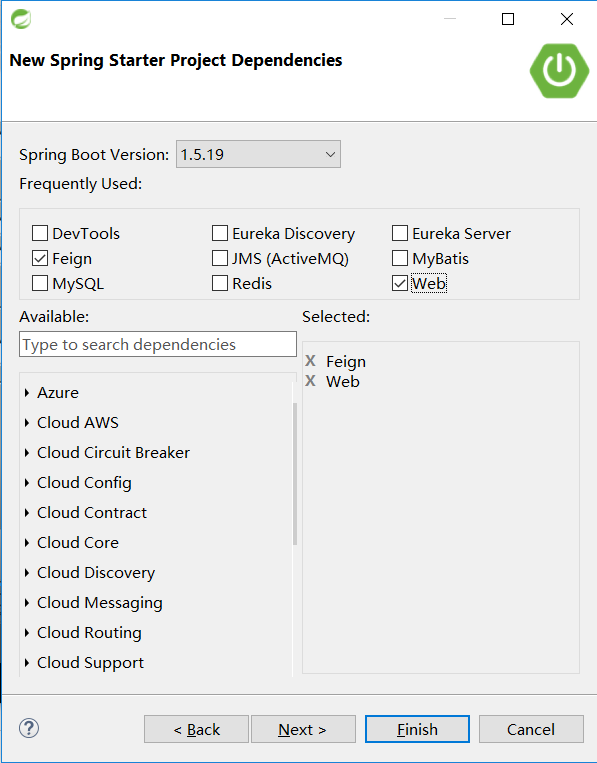
# feign、

里面依赖了hystrix和ribbon

## helloword

### user-service

### teacher-server



yml

|  |
| --- |
| server:  port: 8090    eureka:  client:  service-url:  defaultZone: http://localhost:8761/eureka/  spring:  application:  name: teacher-service |

启动

|  |
| --- |
| @EnableFeignClients  @SpringBootApplication 使用该注解启用feign  **public** **class** TeacherService1Application {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(TeacherService1Application.**class**, args);  }  } |

UserFeign

|  |
| --- |
| **import** com.sxt.cloud.domain.User;  @FeignClient(name="user-service")//serviceId:需要调用服务的名称，spring.application.name相当于duboo的服务的应用名称  **public** **interface** UserFeign {    //1、获取一个用户  @GetMapping("/user/{id}") //user-service/user/{id}  User getUser(@PathVariable("id")Integer id);    //2、获取全部用户，多个参数时，将无法启动，(method have too many menthodbody)  //且必须加@RequestParam(value="page"):反射是获取方法的形式参数，获取到后，并不知道方法的变量的名称。  @GetMapping("users")  **public** List<User> getUserList(@RequestParam(value="page",defaultValue="1")Integer page,@RequestParam(value="size",defaultValue="10")Integer size);  //3、删除，因为是一个参数，不需要加@RequestParam  @DeleteMapping("user/{id}")  **public** String deleteUser(@PathVariable("id") Integer id);    //4、更新  @PostMapping("/user")  **public** String updateUser(@RequestBody User user);  //5 get是否能够将userJson带过去,feign自动的将get请求变为post请求  @GetMapping("/users")  **public** User addUserError(@RequestBody User user);  } |

TeacherController

|  |
| --- |
| @RestController  **public** **class** TeacherController {  @Autowired  **private** UserFeign userFeign;    @GetMapping("/rpc")  **public** String rpc() {  //之前使用restTemplage发送请求,现在使用接口userFeign  //接口能多调用，肯定有代理对象。  User user = userFeign.getUser(1);  System.***out***.println(userFeign);  System.***out***.println(user);  **return** "调用成功";  }  } |

返回结果: 调用成功

通过接口调用user-service，将user-feigin提取为一个接口，让userControler实现这个接口,也就是从controller中方法名，方法参数，返回值，都和接口相同。

接口注意点：

@PathVariable

多参数是要加@requestParam(name=“””)

get请求将json带过去,feign自动的将get请求变为post请求

# Hystrix

## Hystrix

### Hystrix

在分布式系统中，每个服务都可能会调用很多其他服务，被调用的那些服务就是依赖服务，有的时候某些依赖服务出现故障也是很正常的。

Hystrix 可以让我们在分布式系统中对服务间的调用进行控制，加入一些调用延迟或者依赖故障的容错机制。

Hystrix 通过将依赖服务进行资源隔离，进而阻止某个依赖服务出现故障时在整个系统所有的依赖服务调用中进行蔓延；

Hystrix 还提供故障时的 fallback 降级机制。

总而言之，Hystrix 通过这些方法帮助我们提升分布式系统的可用性和稳定性。

### Hystrix 的历史

Hystrix 是高可用性保障的一个框架。Netflix（可以认为是国外的优酷或者爱奇艺之类的视频网站）的 API 团队从 2011 年开始做一些提升系统可用性和稳定性的工作，Hystrix 就是从那时候开始发展出来的。

在 2012 年的时候，Hystrix 就变得比较成熟和稳定了，Netflix 中，除了 API 团队以外，很多其他的团队都开始使用 Hystrix。

时至今日，Netflix 中每天都有数十亿次的服务间调用，通过 Hystrix 框架在进行，而 Hystrix 也帮助 Netflix 网站提升了整体的可用性和稳定性。

2018 年 11 月，Hystrix 在其 Github 主页宣布，不再开放新功能，推荐开发者使用其他仍然活跃的开源项目。维护模式的转变绝不意味着 Hystrix 不再有价值。相反，Hystrix 激发了很多伟大的想法和项目，我们高可用的这一块知识还是会针对 Hystrix 进行讲解。

### Hystrix 的设计原则

1、对依赖服务调用时出现的调用延迟和调用失败进行控制和容错保护。

2、在复杂的分布式系统中，阻止某一个依赖服务的故障在整个系统中蔓延。比如某一个服务故障了，导致其它服务也跟着故障。

3、提供 fail-fast（快速失败）和快速恢复的支持。

4、提供 fallback 优雅降级的支持。

5、支持近实时的监控、报警以及运维操作。

### Hystrix 更加细节的设计原则

阻止任何一个依赖服务耗尽所有的资源，比如 tomcat 中的所有线程资源。

避免请求排队和积压，采用限流和 fail fast 来控制故障。

提供 fallback 降级机制来应对故障。

使用资源隔离技术，比如 bulkhead（舱壁隔离技术）、swimlane（泳道技术）、circuit breaker（断路技术）来限制任何一个依赖服务的故障的影响。

通过近实时的统计/监控/报警功能，来提高故障发现的速度。

通过近实时的属性和配置热修改功能，来提高故障处理和恢复的速度。

保护依赖服务调用的所有故障情况，而不仅仅只是网络故障情况。

### Hynstrix

(1) 通过hystrixCommand或者HystrixObservableCommand来封装对外部依赖的访问请求，这个访问请求一般会运行在独立的线程中

(2) 对于超出我们设定的阈值服务调用，直接进行超时返回，不允许它长时间的阻塞

(3) 对每一个依赖服务进行资源隔离。通过线程池或者是semaphore这两种方式

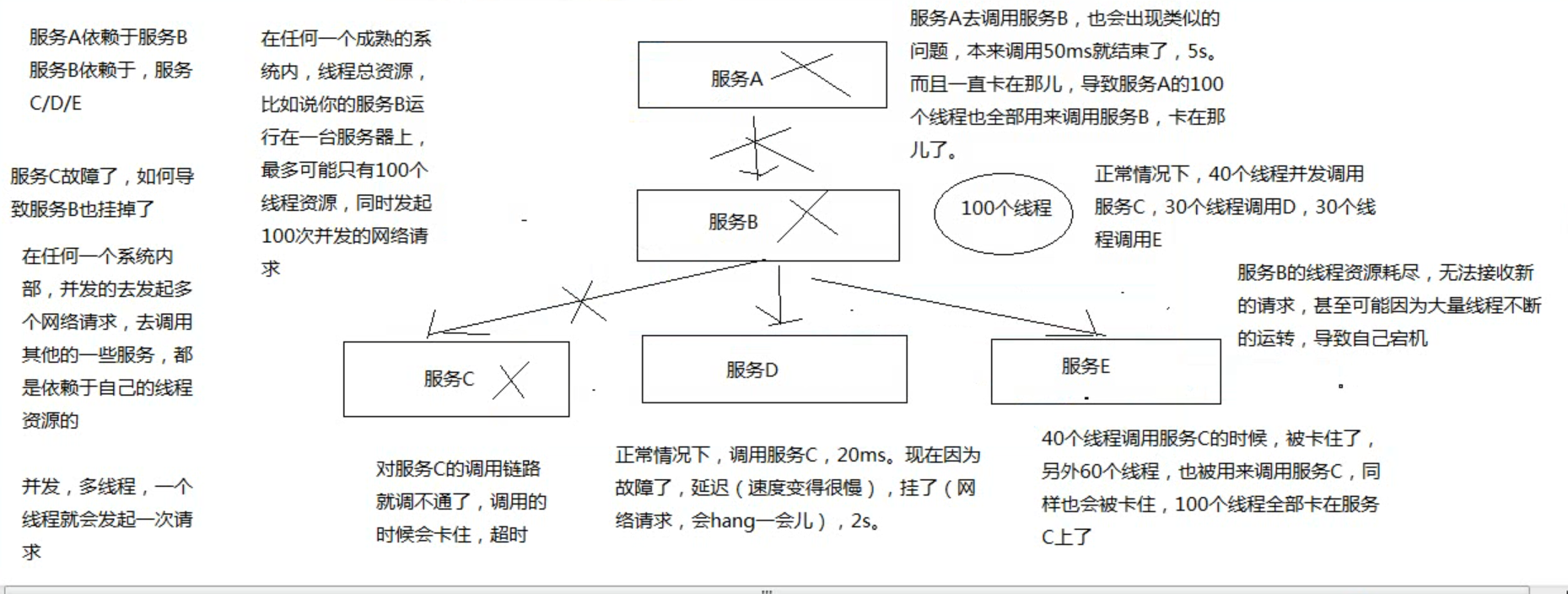
(4) 对依赖服务被调用的成功次数，失败次数，拒绝次数，超时次数进行统计

(5) 如果对某一个依赖服务的调用失败次数超过了一点的阈值，Hystrix自动进行熔断，并在一段时间内对该服务的调用直接进行降级，一段时间后再自动尝试恢复

(6) 当对一个服务调用出现失败、被拒绝、超时、短路等异常情况时，自动调用fallback降级机制

(7) 对属性和配置的修改提供近实时的支持

### 产生的问题



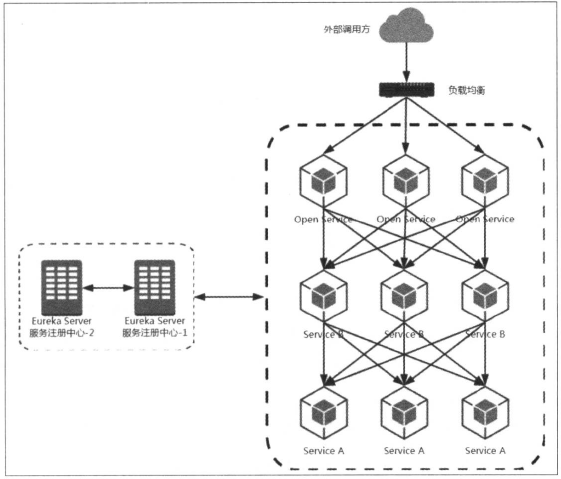
## 原理

### 资源隔离



# zull

### zull引入



首先,我们从运维人员的角度来看看,他们平时都需要做一些什么工作来支持这样的

架构。当客户端应用单击某个功能的时候往往会发出一些对微服务获取资源的请求到后端,这些请求通过F5, Nginx等设施的路由和负载均衡分配后,被转发到各个不同的服务实例上。而为了让这些设施能够正确路由与分发请求,运维人员需要手工维护这些路由规则与服务实例列表,当有实例增减或是IP地址变动等情况发生的时候,也需要手工地去同步修改这些信息以保持实例信息与中间件配置内容的一致性。在系统规模不大的时候,维护这些信息的工作还不会太过复杂,但是如果当系统规模不断增大,那么这些看似简单的维护任务会变得越来越难,并且出现配置错误的概率也会逐渐增加。很显然,这样的做法并不可取,所以我们需要一套机制来有效降低维护路由规则与服务实例列表的难度。

### zuul的功能

动态路由:发现eureka的注册服务名，通过zuul将请求映射到不同的服务

过滤器的实现：1、前端所有的请求都经过zuul，可以做访问权限的验证。

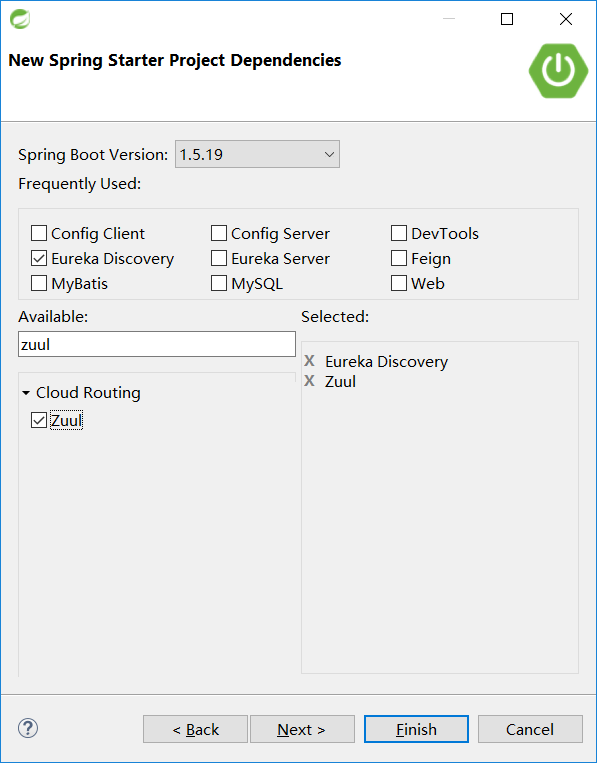
2、流量的限制，流量销峰

统一的授权

可以做方法的调用监控

## zuul的helloworld

### 新建spngboot项目



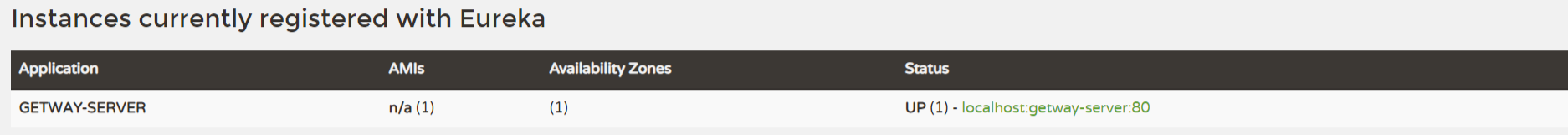
### 修改配置文件

|  |
| --- |
| server:  port: 80  spring:  application:  name: getway-server  eureka:  client:  service-url:  defaultZone: http://localhost:8761/eureka/  #zull的配置  zuul:  routes: #map  user-service-router: #路由名  serviceId: user-service #服务的名字  path: /user-service/\*\* #访问该路径，就进serviceId 对应的微服务  teacher-service-router:  serviceId: teacher-service  path: /teacher-service/\*\* |

### 添加注解

|  |
| --- |
| @SpringBootApplication  @EnableZuulProxy //添加注解  **public** **class** GetwayServerApplication {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(GetwayServerApplication.**class**, args);  }  } |

### 启动



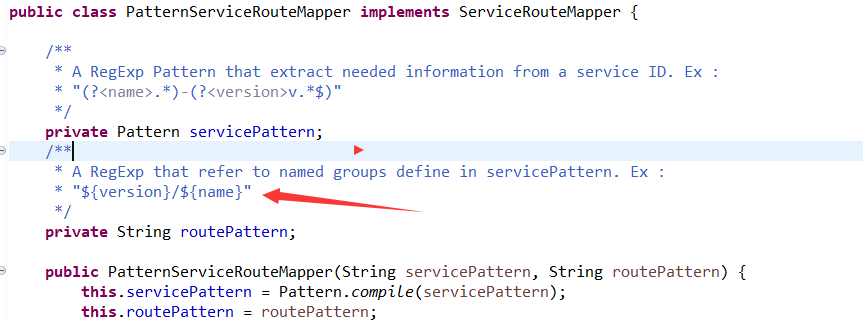
### 测试路由功能

通过访问<http://localhost/user-service/user/1> ，可以访问到微服务。

缺点：配置文件中如果有多个服务，需要些多个。

### 动态路由

删除正则表达式中的版本号。



|  |
| --- |
| @SpringBootApplication  @EnableZuulProxy //添加注解  **public** **class** GetwayServerApplication {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(GetwayServerApplication.**class**, args);  }    //服务路径映射，将请求的服务映射到ureak中的服务。  @Bean  **public** PatternServiceRouteMapper patternServiceRouteMapper() {  //正则表达式  **return** **new** PatternServiceRouteMapper("(?<name>.\*)", "${name}");  }    } |

测试：将配置文件中的关于zull的配置删掉即可

访问://localhost:80/user-service/user 通过访问网关，然后找到eureka中的服务，服务的上线和下线都看的到。

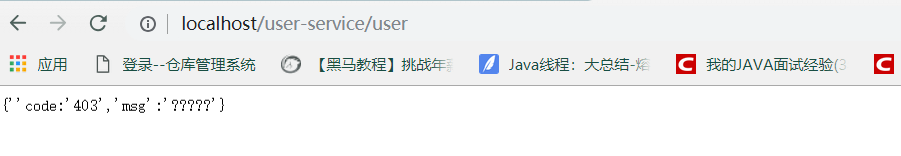
## zuul的过滤器

1、前端所有的请求都经过zuul，可以做访问权限的验证。

2、流量的限制{重要}

### token的拦截

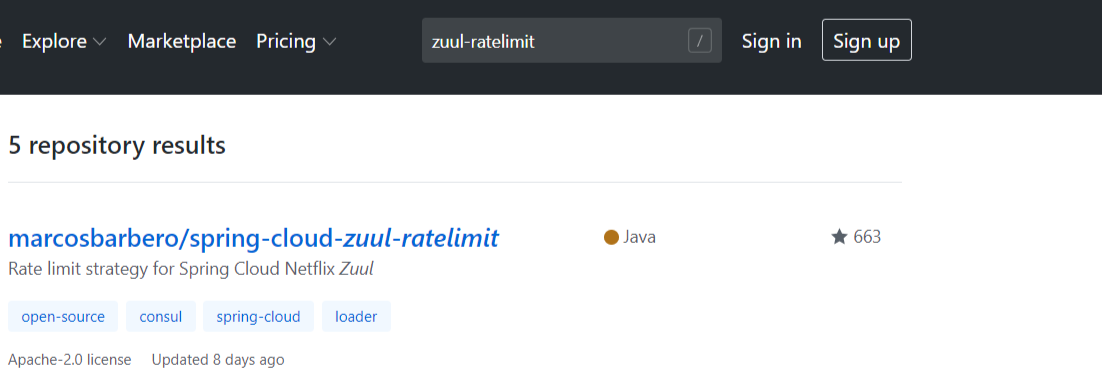
|  |
| --- |
| @Component  **public** **class** TokenFilter **extends** ZuulFilter{    //是否启用过滤器  @Override  **public** **boolean** shouldFilter() {  **return** **true**;  }  //请求并过滤下来了，做业务逻辑，token验证  @Override  **public** Object run() **throws** ZuulException {  //zuul会将所有的请求数据放到自己的context(容器中，)我们可以获取到该context  RequestContext context = RequestContext.*getCurrentContext*();  HttpServletRequest request = context.getRequest();  String token = request.getParameter("toekn");  //哦安短token是否存在  **if**(token==**null**) {//提醒用户登录  //设置响应内容JSON格式,不能通过context获取response  context.setResponseBody("{''code:'403','msg':'用户未登录'}");  //设置，没有通过zuul  context.setSendZuulResponse(**false**);  **return** **null**;  }  **if**(!token.equals("123456")) {//token验证错误  //设置响应内容JSON格式,不能通过context获取response  context.setResponseBody("{''code:'403','msg':'用户token错误'}");  //设置，没有通过zuul  context.setSendZuulResponse(**false**);  **return** **null**;  }  //token验证通过  context.setSendZuulResponse(**true**);  //return 没有意义  **return** **null**;  }    //过滤器类型  @Override  **public** String filterType() {  //pre ：请求刚进zull  //router:请求将要被路由到微服务  //post 请求已经访问玩微服务了  //error 请求在访问微服务发送错误，  **return** "pre";  }  //filter的执行顺序，数子越小，越先被执行  @Override  **public** **int** filterOrder() {  **return** 0;  }  } |



## 限流

此方式也是使用的zuulfilter

导入startter



### 导入依赖

|  |
| --- |
| <dependency>  <groupId>com.marcosbarbero.cloud</groupId>  <artifactId>spring-cloud-zuul-ratelimit</artifactId>  <version>LATEST</version>  </dependency> |

### 配置文件

|  |
| --- |
| zuul:  ratelimit:  enabled: **true**  behind-proxy: **true**  default-policy-list: #全局的设置  - limit: 10 # 最多访问的次数optional - request number limit per refresh interval window  quota: 1000 #访问的时间间隔10optional - request time limit per refresh interval window (in seconds)  refresh-interval: 60 # 60 s 访问的总数量为10 default value (in seconds)  type: #optional  - origin |

### 测试

连续访问一个请求



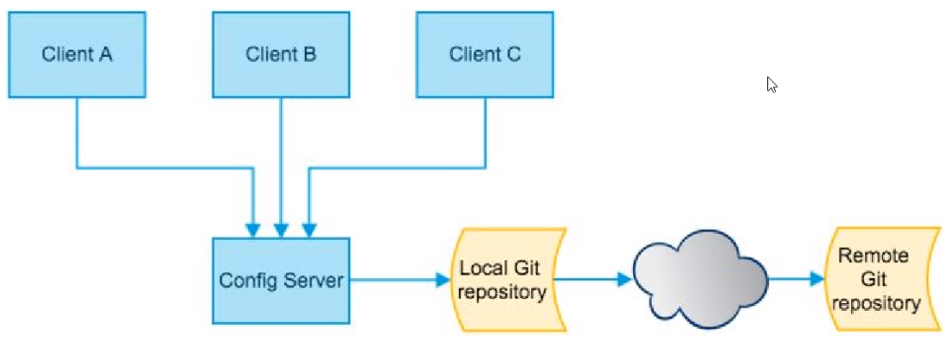
# 分布式配置中心(Config)

## spring cloud config

### 引入

微服务意味着要将单体应用中的业务拆分成一个个子服务，每个服务的粒度相对较小，因此系统中会出现大量的服务。由于每个服务都需要必要的配置信息才能运行，所以一套集中式的、动态的配置管理设施是必不可少的。SpringCloud提供了ConfigServer来解决这个问题，我们每一个微服务自己带着一个application.yml，上百个配置文件的管理。

### Spring Cloud Config



Spring Cloud Config：集中是管理配置文件

SpringCloud Config为微服务架构中的微服务提供集中化的外部配置支持，配置服务器为各个不同微服务应用的所有环境提供了一个中心化的外部配置。

### 组成

SpringCloud Config分为服务端和客户端两部分。

服务端也称分布式配置中心，它是一个独立的微服务应用，用来连接配置服务器并为客户端提供获取配置信息，加密/解密信息等访问接口

客户端则是通过指定的配置中心来管理应用资源，以及与业务相关的配置内容，并在启动的时候从配置中心获取和加载配置信息配置服务器默认采用git来存储配置信息，这样就有助于对环境配置进行版本管理，并且可以通过git客户端工具来方便的管理和访问配置内容。

### 解决问题

1、集中管理配置文件

2、不同环境不同配置，动态化的配置更新，分环境部署比如dev/test/prod/beta/releasel

3、运行期间动态调整配置，不再需要在每个服务部署的机器上编写配置文件，服务会向配置中心统一拉取配置自己的信息

4、当配置发生变动时，服务不需要重启即可感知到配置的变化并应用新的配置

5、将配置信息以REST接口的形式暴露

### 思路

实现目的：配置文件共享，达到服务的一致性。

实现思路：

1、各个服务共同拥有一个共有的配置文件作为公有配置。

2、没个服务还有自己特有的配置文件。

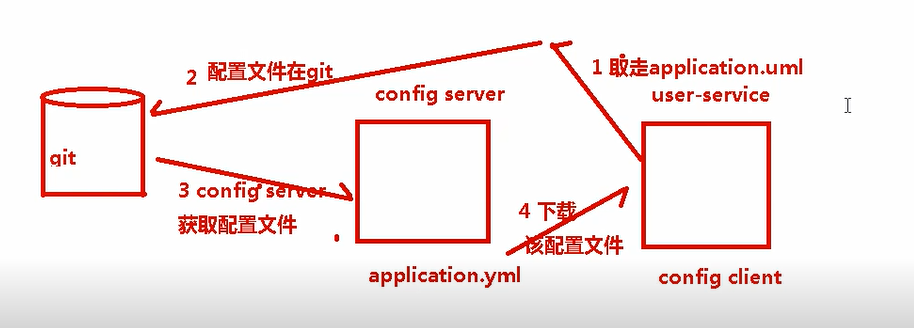
携程的阿波罗项目也可以实现

spring cloud config如何实现：

git：配置文件存放到git中，修改后，config server会自动拉取配置文件。

config server：

client ：客户client与微服务集成在一起。拉取git的配置文件。



## configServer

### github操作

1、用自己的GitHub账号在GitHub上新建一个名为microservicecloud-config

2、由上一步获得SSH协议的git地址：

<https://github.com/Padingpading/microservicecloud-config.git>

3、本地硬盘目录上新建git仓库并clone。

例如在E:\git\cloudcpnfig：克隆项目

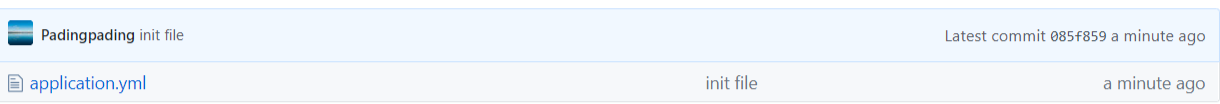
$ git clone <https://github.com/Padingpading/microservicecloud-config.git>

4、在仓库中新建一个application.yml（保证yml的编码为utf8），并将该文件上传到github。

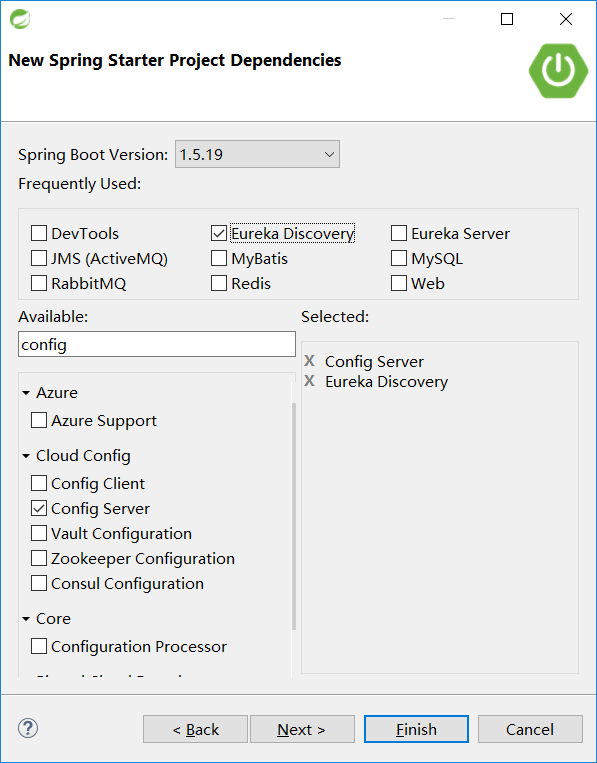
git add application.yml

git commit application.yml -m “新增”

git push



### 新建项目



### 配置文件

|  |
| --- |
| #1、添加端口,默认为8888  server:  port: 8888  #2、应用名  spring:    cloud:  config:  server:  git:  #git仓库位置  uri: https://gitee.com/Padingpading/test-spring-cloud-config.git  username: 449196538@qq.com  password: www.libin.com0  #git仓库下搜索位置  search-paths:  - /config  #3、注册中心  eureka:  client:  service-url:  default: http://localhost:8761/eureka |

### 添加注解

|  |
| --- |
| @SpringBootApplication  @EnableConfigServer  **public** **class** ConfigServerApplication {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(ConfigServerApplication.**class**, args);  }  } |

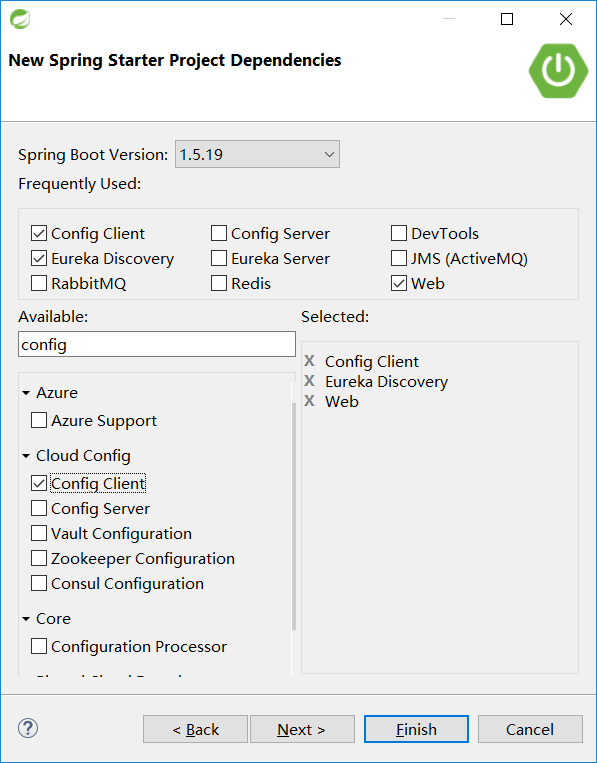
### 启动测试

访问文件：<http://localhost:8888/user-service.yml> ，会自动从云端拉去。

如果在云端修改了文件，会自动拉去。

## 客户端client

### 新建项目、



### 配置文件

gitee中的配置：一些公共的配置

|  |
| --- |
| #3、注册中心  eureka:  client:  service-url:  default: http://localhost:8761/eureka  city:  name: beijin |

1、修改配置文件的名称，目的是增加本地文件的优先级，spring会优先读取bootstrap.yml。

|  |
| --- |
| #1、配置config服务器的地址,默认为8888，目的是去拉取文件的位置  spring:  cloud:  config:  uri: http://localhost:8888  #2、要拉取的文件,spring会拉取与applicationname相同名字的文件  application:  name: config-client |

### 测试

|  |
| --- |
| @Value("${city.name}")  **private** String cityName;  @Test  **public** **void** contextLoads() {  System.***out***.println(cityName);  } |

结果：客户获取到cityName的值为beijing

### 动态刷新

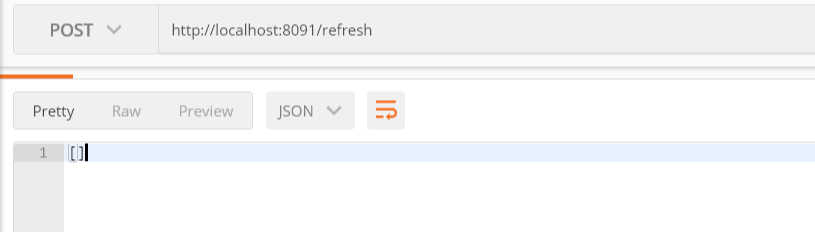
添加依赖

|  |
| --- |
| <dependency>  <!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-actuator -->  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-actuator</artifactId>  </dependency> |

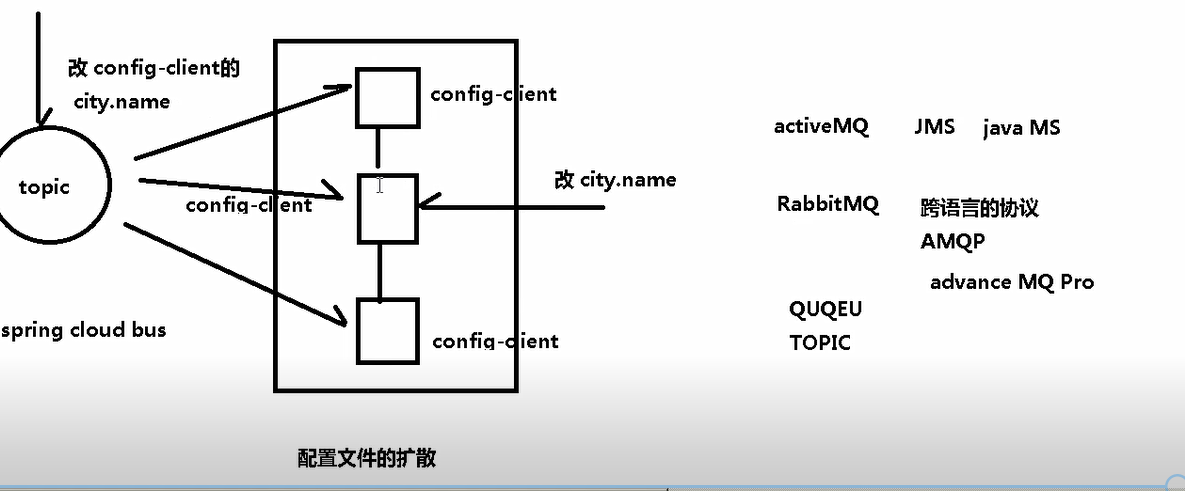
修改配置文件

|  |
| --- |
| #动态刷新  management:  security:  enabled: **false** |

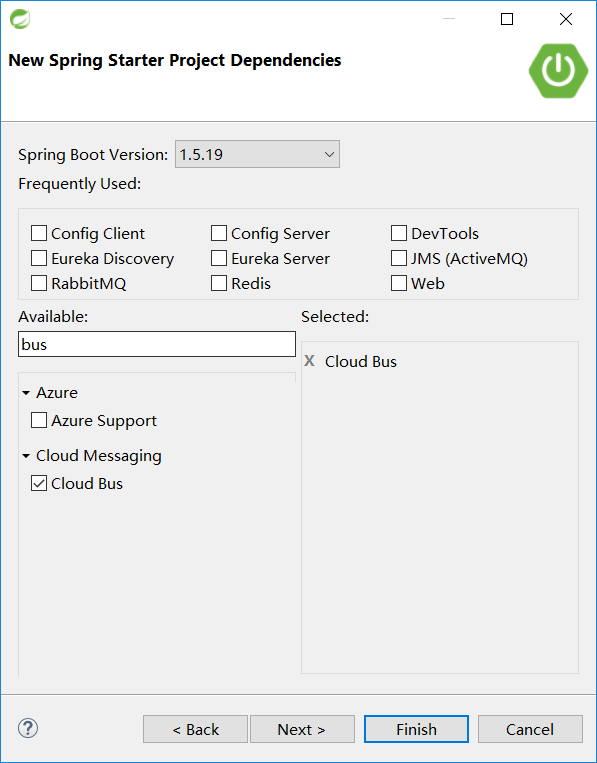
在git 修改配置信息，然后使用postman 使用refresh



## 配置文件的扩散



### 新建项目



### 依赖

|  |
| --- |
| <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-bus-amqp</artifactId>  </dependency> |

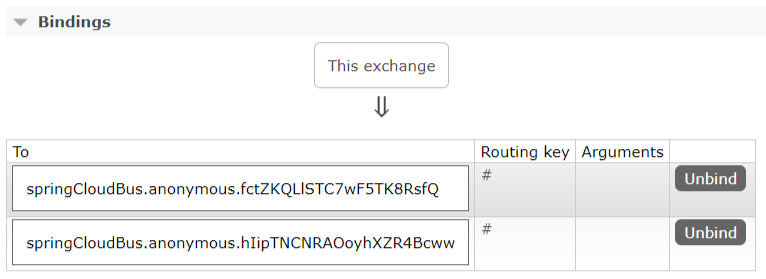
### 配置

|  |
| --- |
| server:  port: 8081  spring:  rabbitmq:  host: 192.168.10.136  port: 5672  username: user  password: 123456  下面的不添加也可以  cloud:  bus:  enabled: **true**  env: #spring的enviroment类  enabled: **true**  refresh:  enabled: **true** |

### 效果

发个消息，就都能收到这个消息，我看见是让我刷新配置文件和对象的一个消息，

我就执行刷新。



### client端配置文件和依赖

配置文件

|  |
| --- |
| spring:  rabbitmq:  host: 192.168.10.136  port: 5672  username: user  password: 123456 |

依赖

|  |
| --- |
| <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-bus</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-bus-amqp</artifactId>  </dependency>  <dependency> |

客户端刷新

<http://localhost:8091/refresh> 刷新单个

http://localhost:8091//bus/refresh 刷新集群

## config-server集群

### config-server

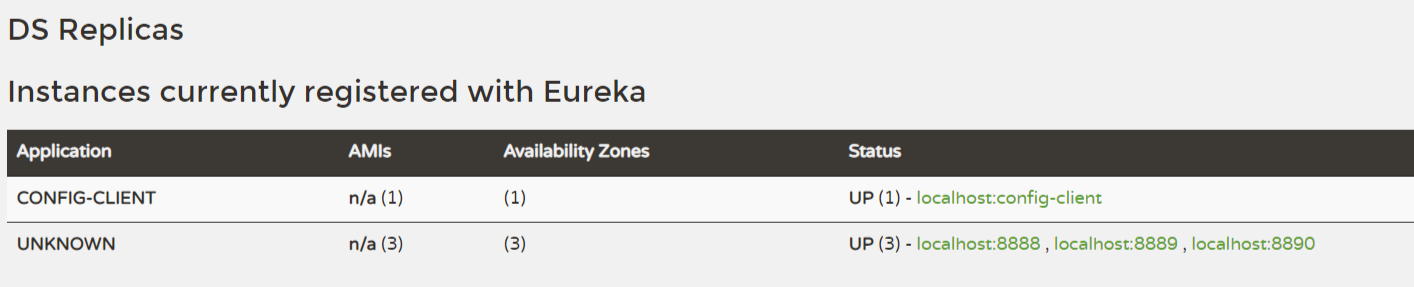
启动config-server，以不同的端口启动

### config-client

配置

|  |
| --- |
| #1、将服务注册到服务中心  eureka:  client:  service-url:  defaultZone: http://localhost:8761/eureka  #1、配置config服务器的地址,默认为8888，目的是去拉取文件的位置  spring:  cloud:  config:  discovery:  enabled: **true** #开启客户端的服务查找  service-id: config-server #查找的服务名，也就是集群的名字  # uri: http://localhost:8888 配置服务器的地址，单机版要写死    #2、要拉取的文件,spring会拉取与applicationname相同名字的文件  application:  name: config-client |

### 结果



config-client:会在服务中心任意一个服务拉去

## 切换配置文件

### gitee上的是三个文件

gitee分别有是哪个不同版本的文件，先把它拉到，本地的config-server中。



### service的配置文件

在service的配置文件，配置要切换的模式，service会智能拉取，service-环境.yml 对应的yml文件。

|  |
| --- |
| #1、将服务注册到服务中心  eureka:  client:  service-url:  defaultZone: http://localhost:8761/eureka  #1、配置config服务器的地址,默认为8888，目的是去拉取文件的位置  spring:  cloud:  config:  discovery:  enabled: **true** #开启客户端的服务查找  service-id: config-server #查找的服务名，也就是集群的名字  profile: dev #切换到开发环境 ，pro 为开发环境，test 为测试环境  # uri: http://localhost:8888 配置服务器的地址，单机版要写死    #2、要拉取的文件,spring会拉取与applicationname相同名字的文件  application:  name: config-client |

# spring boot admin

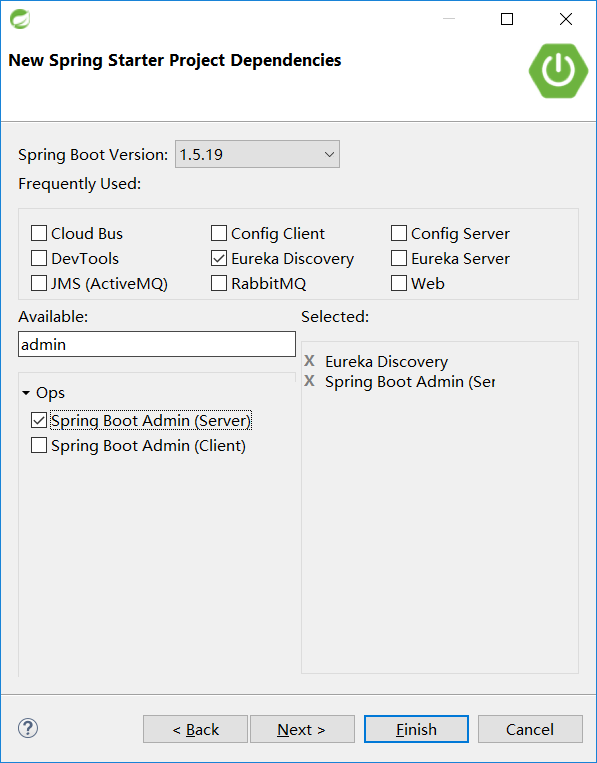
## spring boot admin介绍

spring boot admin是一个基于eureka的监控中心。



## admin server端

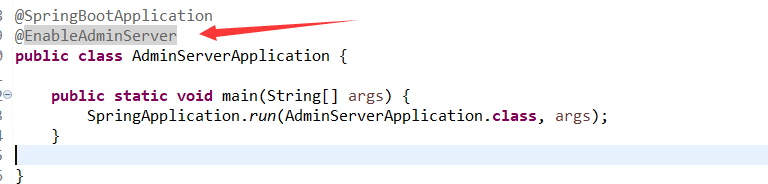
### 创建项目



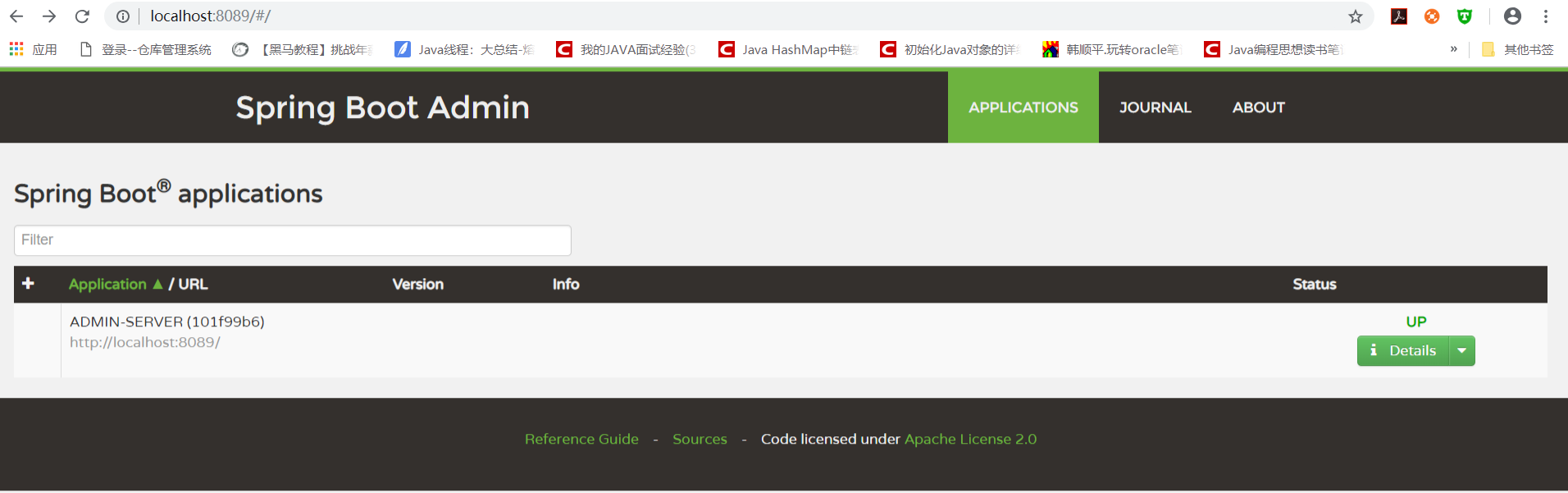
### 修改配置文件

|  |
| --- |
| server:  port: 8089  spring:  application:  name: admin-server  eureka:  client:  service-url:  defaultZone: http://localhost:8761/eureka |

### 添加注解

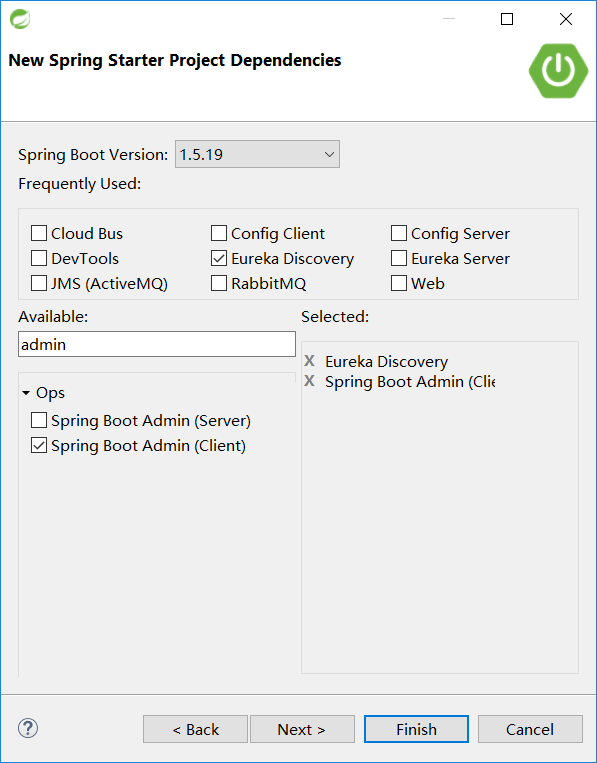


### 效果



## client端(与service集成)

### 新建admin-client

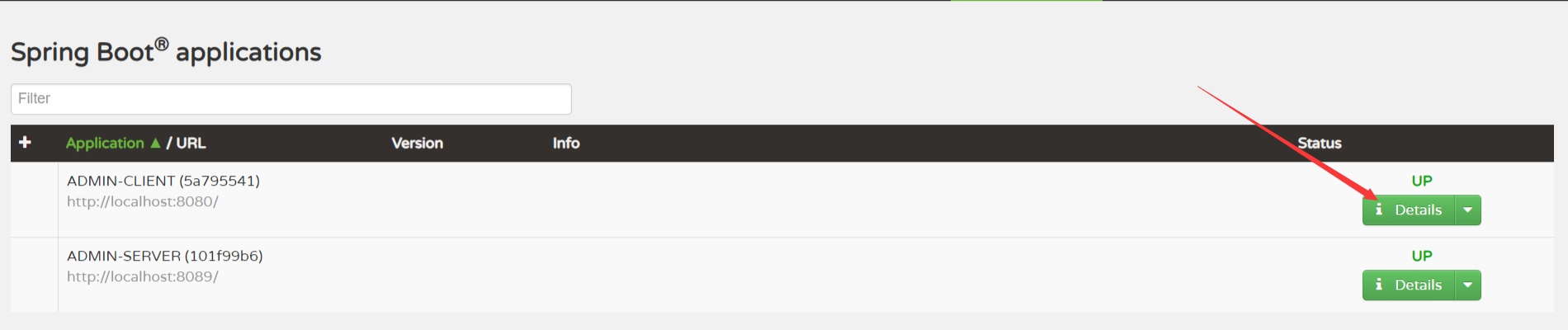


### 修改配置文件

|  |
| --- |
| server:  port: 8080  spring:  application:  name: admin-client  boot:  admin:  url:  - http://localhost:8098  eureka:  client:  service-url:  defaultZone: http://localhost:8761/eureka  management:  security:  enabled: **false** |

### 启动

## 启动测试



# spring boot sercurity

## spring sercurity

### 对比shiro

验证:登录

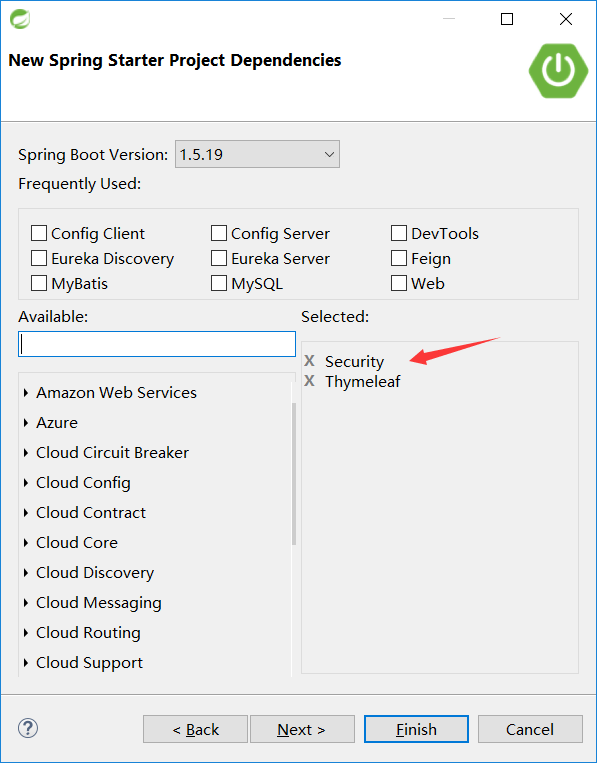
授权：权限的判断

shiro:使用简单，在分布式里面，拓展性很差，shrio的sesison没有共享。

security：spring天然集成，在分布式项目里面，可以完美的集成里面

## helloworld(验证和授权)

### 新建项目



### 选择依赖

添加依赖spring和securit 的标签支持

|  |
| --- |
|  |

### 修改配置文件

|  |
| --- |
| spring:  thymeleaf:  cache: **false**  encoding: UTF-8  mode: HTML5 |

### 修改安全策略

定义验证和授权的功能

|  |
| --- |
| @Configuration  @EnableGlobalMethodSecurity(prePostEnabled=**true**)//以后我们权限验证级别都是对于controller的方法的验证  **public** **class** SecurityConfig **extends** WebSecurityConfigurerAdapter{  //定义一个默认的用户,定义之后用户可以登录成功  //  @Override  **protected** **void** configure(AuthenticationManagerBuilder auth) **throws** Exception {  auth.inMemoryAuthentication().withUser("sd").password("123456").roles("USER").  and().withUser("ls").password("123456").roles("ADMIN");    }  //定义登录行为和授权行为  @Override  **protected** **void** configure(HttpSecurity http) **throws** Exception {  //匹配"/css/\*\*","/js/\*\*","/img/\*\*"的路径所有人都可以访问  http.authorizeRequests().antMatchers("/css/\*\*","/js/\*\*","/img/\*\*").permitAll();  //定义登录行为,登录成功做什么事情，登录失败做什么事情  http.formLogin().loginPage("/login").  defaultSuccessUrl("/index",**true**).//登录成功后的页面  failureUrl("/login-error").permitAll().//登录失败后的页面  and().exceptionHandling().accessDeniedPage("/403");//没有权限访问抛出异常的处理，访问403请求  //请求的授权匹配角色  //http.authorizeRequests().antMatchers("/user/\*\*").hasRole("USER");  //定义和授权行为:，其他的访问地址，必须授权才能访问  http.authorizeRequests().antMatchers("/\*\*").authenticated();  //禁用跨站的伪造请求  http.csrf().disable();  }  } |

登录请求

|  |
| --- |
| /\*\*登录页面没有角色验证  \* **@author** Padingpading  \*  \*/  @Controller  **public** **class** LoginController {  //隐含的一个post  @GetMapping("/login")  **public** String login() {  //会对用户名和密码进行判断  **return** "login";  }  //登录成功之后跳转的页面  @GetMapping("/index")  **public** String index() {  **return** "index";  }  //登录失败后跳转的页面  @GetMapping("/login-error")  **public** String error(Model model) {  model.addAttribute("isError", **false**);  **return** "login";  }  //抛出异常之后才能访问的请求  @GetMapping("/403")  **public** String deny(Principal user,Model model) {  model.addAttribute("user", user.getName());  **return** "403";  }  } |

权限方法：只有有该权限的用户才能访问。

|  |
| --- |
| @Controller  @RequestMapping("/user")  **public** **class** UserController {  //使用该注解@PreAuthorize做一个方法级别的权限验证，el表达式  //方法级别的拦截,添加之后除非有USER的权限，否则无法访问  @PreAuthorize("hasAuthority('ROLE\_USER')")  @GetMapping("/userindex")  **public** String userIndex() {  **return** "/user/userindex";  }  } |

## 使用数据库查询角色

# 分布式事务

# oauth

## OAuth是什么？

### oauth引入



小新现在想要使用一个“在线打印服务”来打印一些照片，同时小新的照片都存储在了“云网盘”上，按照传统的方式小新要怎么做呢？

1、将照片从“云网盘”上down下来，在上传到“在线打印服务”，然后开始打印。

2、下载/上传太麻烦了，小新可以直接把“云网盘”的账号和密码告诉“在线打印服务”，由“在线打印服务”下载照片在上传。

对于上面的两种方法，方法一太麻烦但是相对于小新来说是安全的；方法二对于小新是比较方便，但是将账号密码告诉了“在线打印服务”就相当于把所有的“云网盘”资料交给了它，这肯定是不可取的。

所以，为了解决类似上面的问题，Oauth协议诞生了。

### OAuth

OAuth 即 Open standard for Authorization

OAuth是一个网络开放协议。为保证用户资源的安全授权提供了简易的标准oauth的好处：

允许用户授权第三方网站或应用，访问用户存储在其它网站上的资源，而不需要将用户名和密码提供给第三方网站或分享他们数据的内容

对于用户:免去了繁琐的注册过程,降低了注册成本,提高了用户体验

对于消费方:简化自身会员系统的同时又能够带来更多的用户和流量。

对于服务提供者:围绕自身进行开发,增加用户粘性

目前oauth和版本是2.0即oauth2.0，而且不向下兼容。本文章主要针对oauth2.0进行讲解

## Oauth2.0协议流程

### 角色

在介绍协议流程之前先要说明一下oauth2.0定义的几个角色：

resource owner：资源所有者，这里可以理解为用户。

client：客户端，可以理解为一个第三方的应用程序。

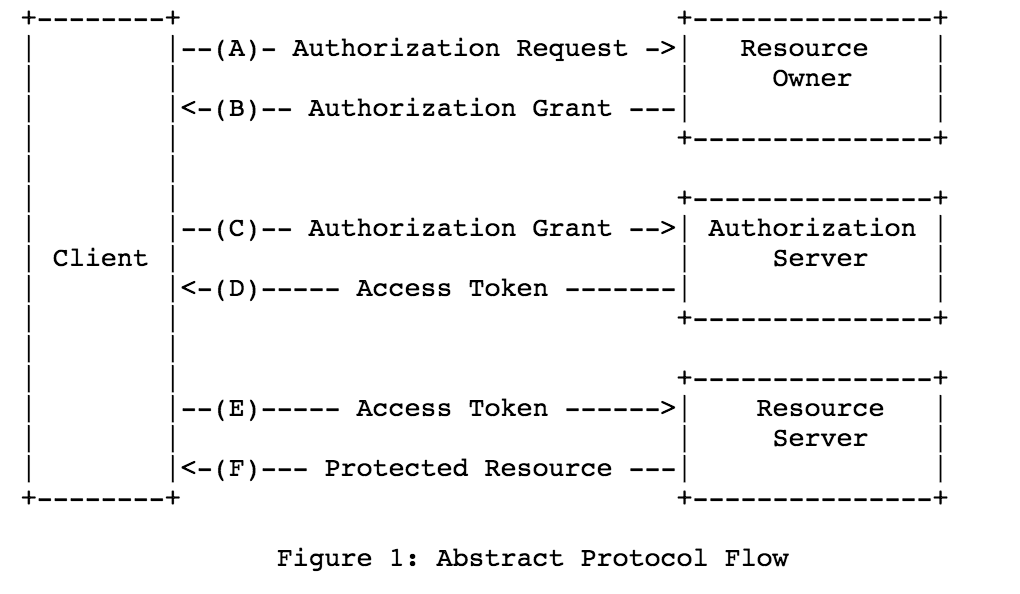
resource server：资源服务器，它存储用户或其它资源。

authorization server：授权服务器，它认证resource owner的身份，为 resource owner提供授权审批流程，并最终颁发授权令牌(Access Token)。

user-agent：用户代理，这里可以理解为“浏览器”。

这里面有个需要注意的地方，这里只是在逻辑上把authorization server与resource server区分开来；在物理上，authorization server与resource server的功能可以由同一个服务器来提供服务。

### 流程



（A）用户打开客户端以后，客户端要求用户给予授权。

（B）用户同意给予客户端授权。

（C）客户端使用上一步获得的授权，向认证服务器申请令牌。

（D）认证服务器对客户端进行认证以后，确认无误，同意发放令牌。

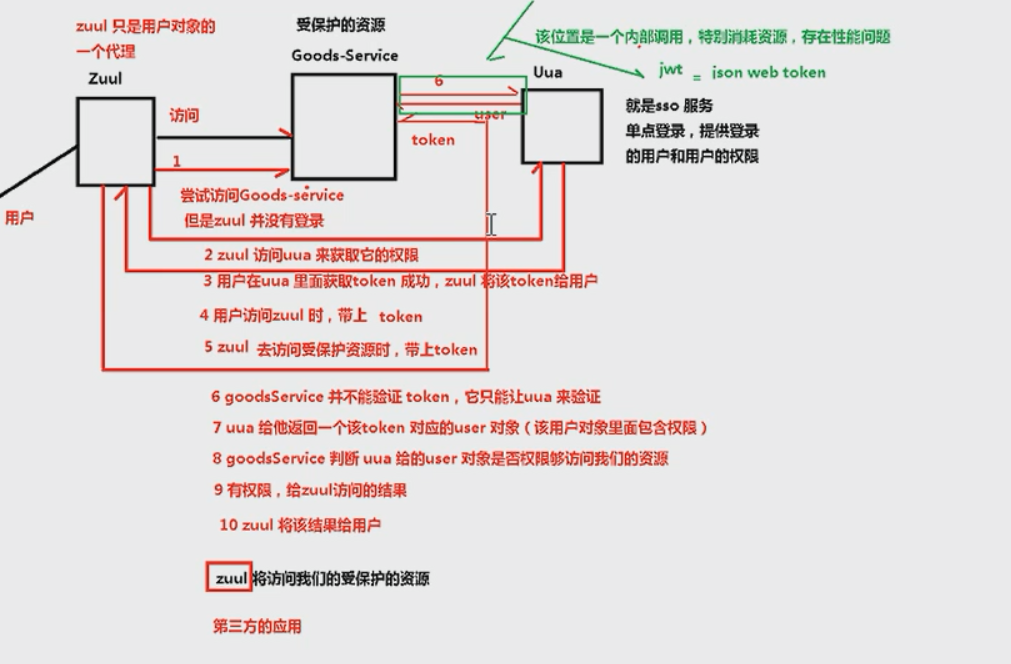
（E）客户端使用令牌，向资源服务器申请获取资源。

（F）资源服务器确认令牌无误，同意向客户端开放资源。

上面六个步骤之中，B是关键，即用户怎样才能给于客户端授权。有了这个授权以后，客户端就可以获取令牌，进而凭令牌获取资源。

下面就介绍一下oauth2.0获取授权的几种方式。

## 微服务的架构体系里面

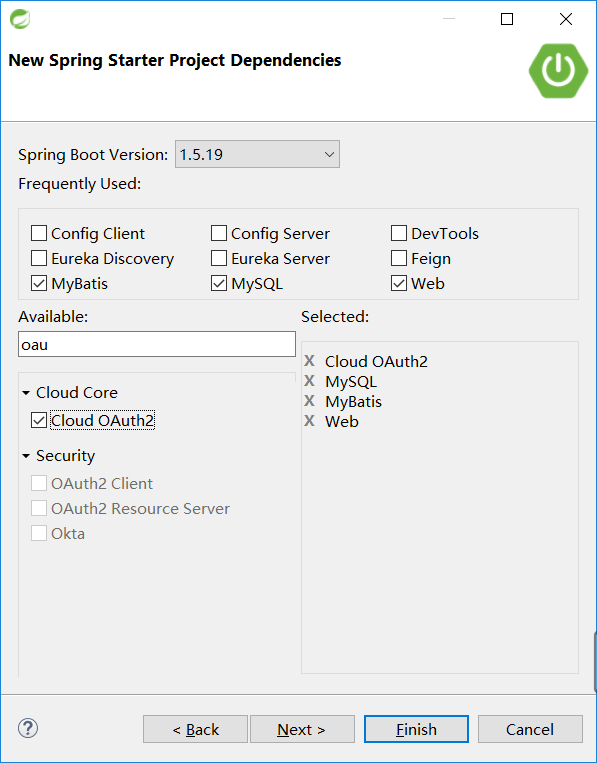


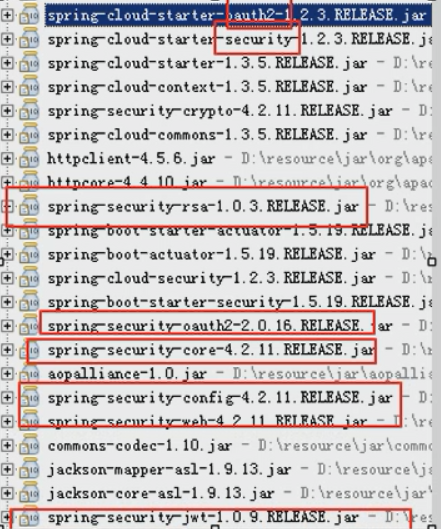
1、构建uua系统

2、将接口资源保护起来

3、

### 构建uua系统





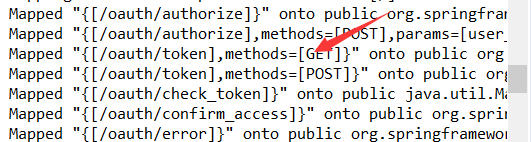
### 配置

yml

|  |
| --- |
| #基于过滤器来做的  server:  port: 8100  spring:  application:  name: uua-service  redis:  host: 192.168.10.136  port: 6379  security:  oauth2:  resource:  filter-order: 3 #过滤器在第三个执行 |

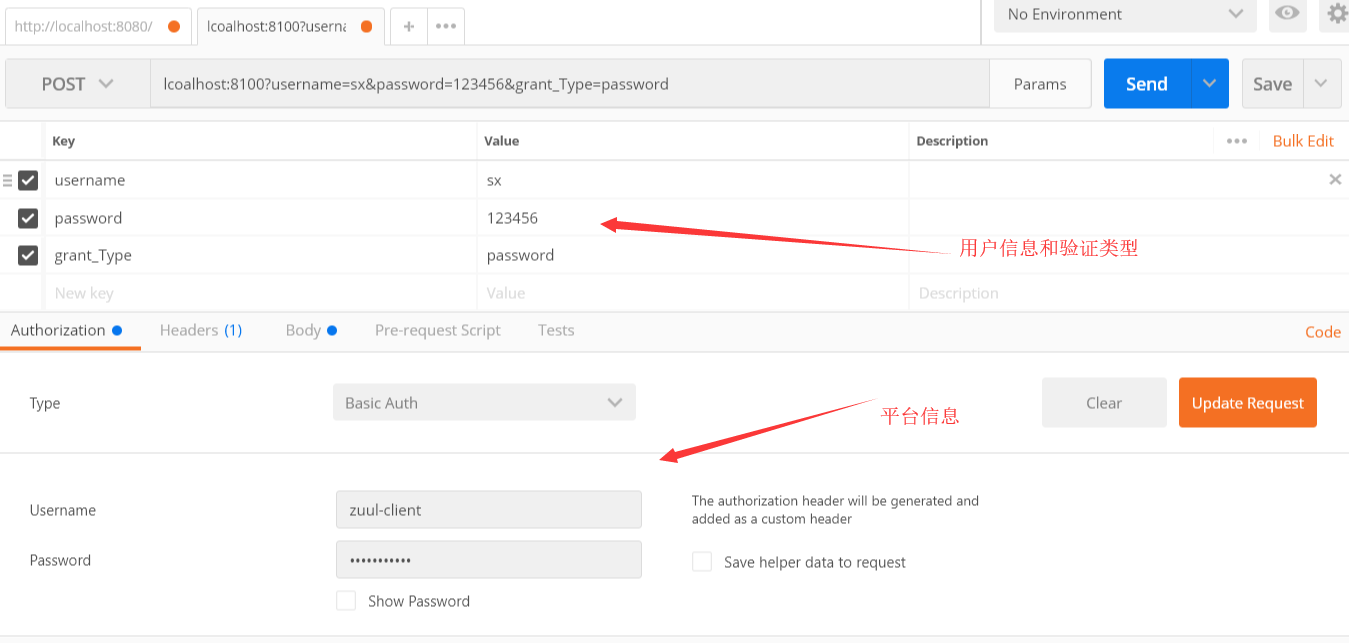
### 验证配置

## 配置类型



### 请求

localhost:8100/oauth/token?username=sx&password=123456&grant\_type=password



### redis中

token的数量



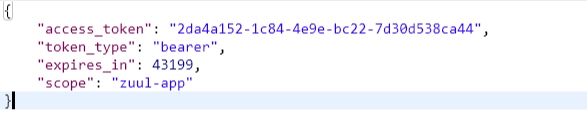
授权的数量



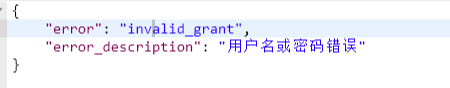
平台的数量



### 得到token



使用错误的账号密码



### 获取用户信息

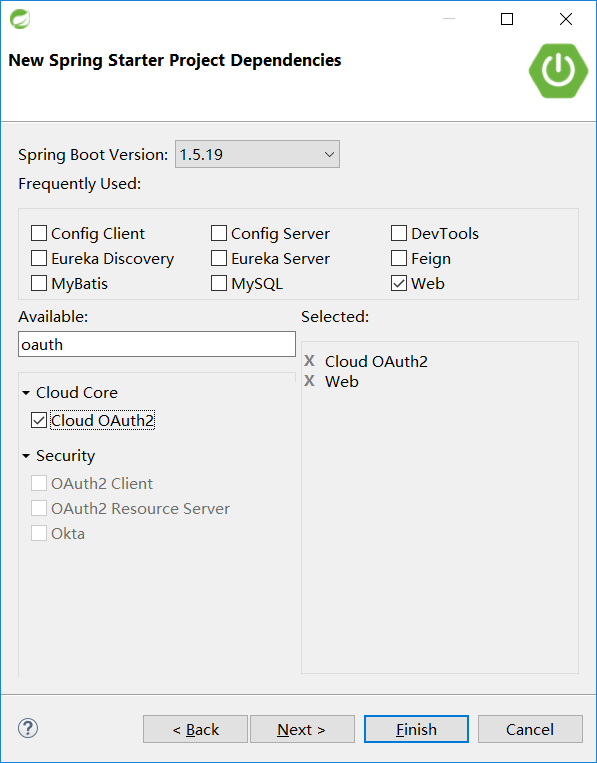
<http://localhost:8100/current/user?access_token=547e9ab1-6438-42c5-9111-5d1a451a1847>

通过token获取用户对象!



### 新建一个资源保护起来

创建资源服务器：受到别人的保护



|  |
| --- |
| security:  oauth2: #第三方应用带token方法问我，我要使用token去uua服务换一个user对象  resource:  user-info-uri: http://localhost:8100/current/user #获取用户对象的地址  filter-order: 3  client:  access-token-uri: http://localhost:8100/oauth/authorize  user-authorization-uri: http://localhost:8100/oauth/token #获取token的url  client-id: zuul-client #平台的id  client-secret: zuul-secret #平台的密码  scope: zuul-app |

新建一个受保护的资源

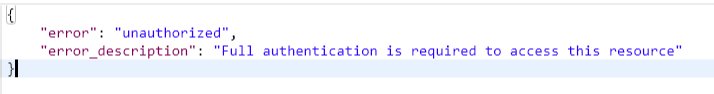
|  |
| --- |
| @Controller  **public** **class** TestController {  @PreAuthorize("hasAuthority('ROLE\_USER')")  @GetMapping("/goods")  @ResponseBody  **public** String goods() {  **return** "我是商品受保护";  }    @PreAuthorize("hasAuthority('ROLE\_ADMIN')")  @DeleteMapping("/goods")  @ResponseBody  **public** String delGoods() {  **return** "我是更高的角色";  }  } |

配置类

|  |
| --- |
| //开启资源的受保护  @EnableResourceServer  @EnableGlobalMethodSecurity(prePostEnabled=**true**)//开启方法级别的拦截  **public** **class** ResourceConfig **extends** ResourceServerConfigurerAdapter{    @Override  **public** **void** configure(HttpSecurity http) **throws** Exception {  //所有的自由都必须受保护  http.authorizeRequests().anyRequest().authenticated();  //  http.csrf().disable();  }  } |

## 验证

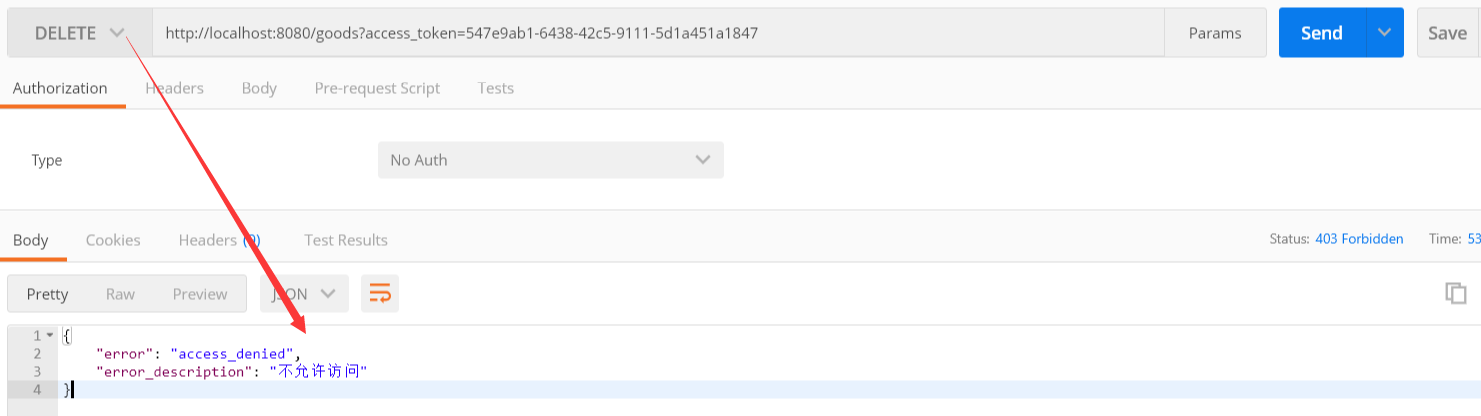
没有开启验证服务器访问时



访问时带上token



访问没有权限的方法



## 使用JWT作为获取的token

### oauth2作为jwt

|  |
| --- |
| @Configuration  @EnableAuthorizationServer  **public** **class** AuthJwtConfig **extends** AuthorizationServerConfigurerAdapter{  @Autowired  **private** AuthenticationManager authenticationManager;    @Override  **public** **void** configure(ClientDetailsServiceConfigurer clients) **throws** Exception {  clients.inMemory().withClient("micr-service").secret("micr-servret").authorizedGrantTypes("password")  .scopes("micr-service-scope");  }  //配置jwt的存储方式  @Override  **public** **void** configure(AuthorizationServerEndpointsConfigurer endpoints) **throws** Exception {    endpoints.tokenStore(tokenStore()).tokenEnhancer( jwtTokenEnhancer()).authenticationManager(authenticationManager);  **super**.configure(endpoints);  }  **public** TokenStore tokenStore() {  **return** **new** JwtTokenStore(jwtTokenEnhancer());  }  **public** JwtAccessTokenConverter jwtTokenEnhancer() {  JwtAccessTokenConverter accessTokenConverter = **new** JwtAccessTokenConverter();  accessTokenConverter.setSigningKey("123");  **return** accessTokenConverter;  }    @Override  **public** **void** configure(AuthorizationServerSecurityConfigurer security) **throws** Exception {  security.tokenKeyAccess("permitAll()").checkTokenAccess("isAuthorizatied()");//是否登录  }  } |

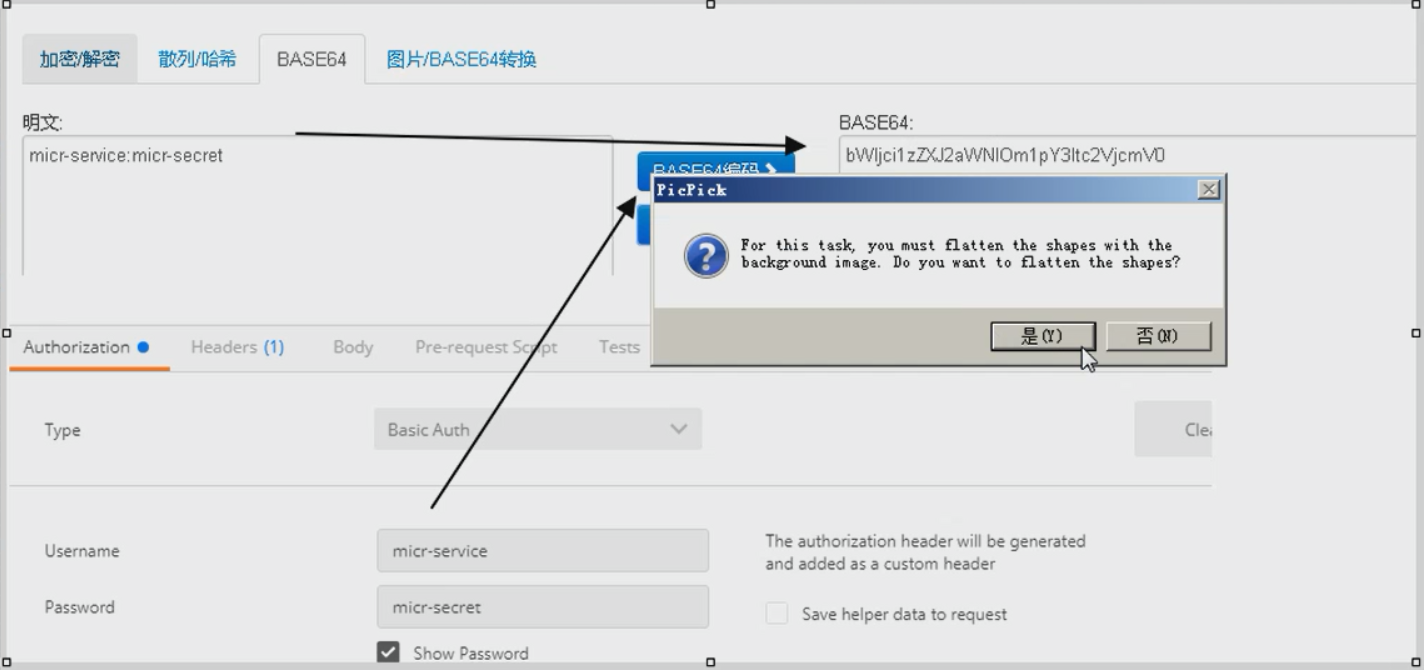
获取token

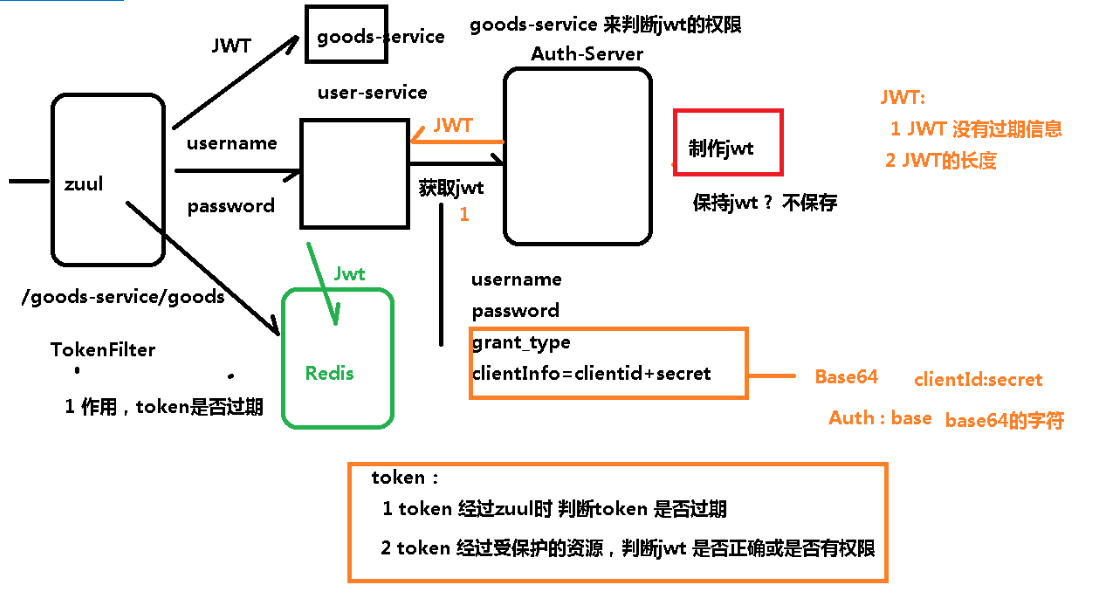


解析的jwt



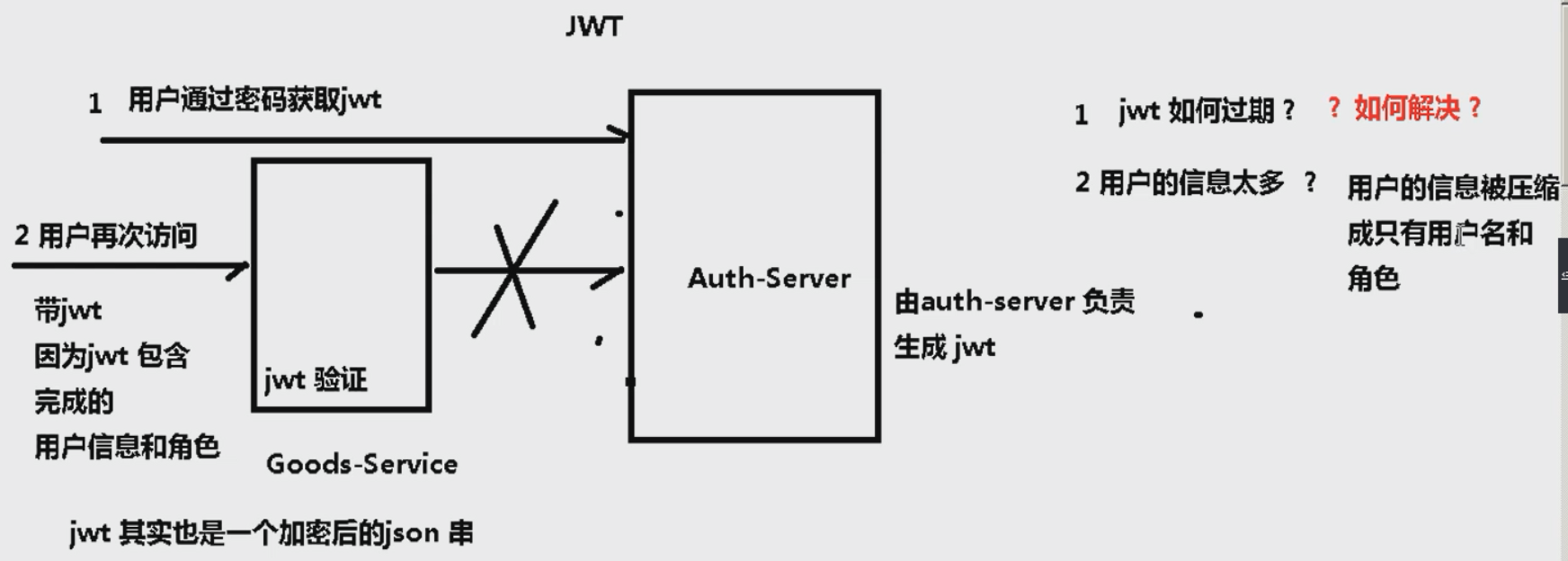
### 使用jwt访问资源

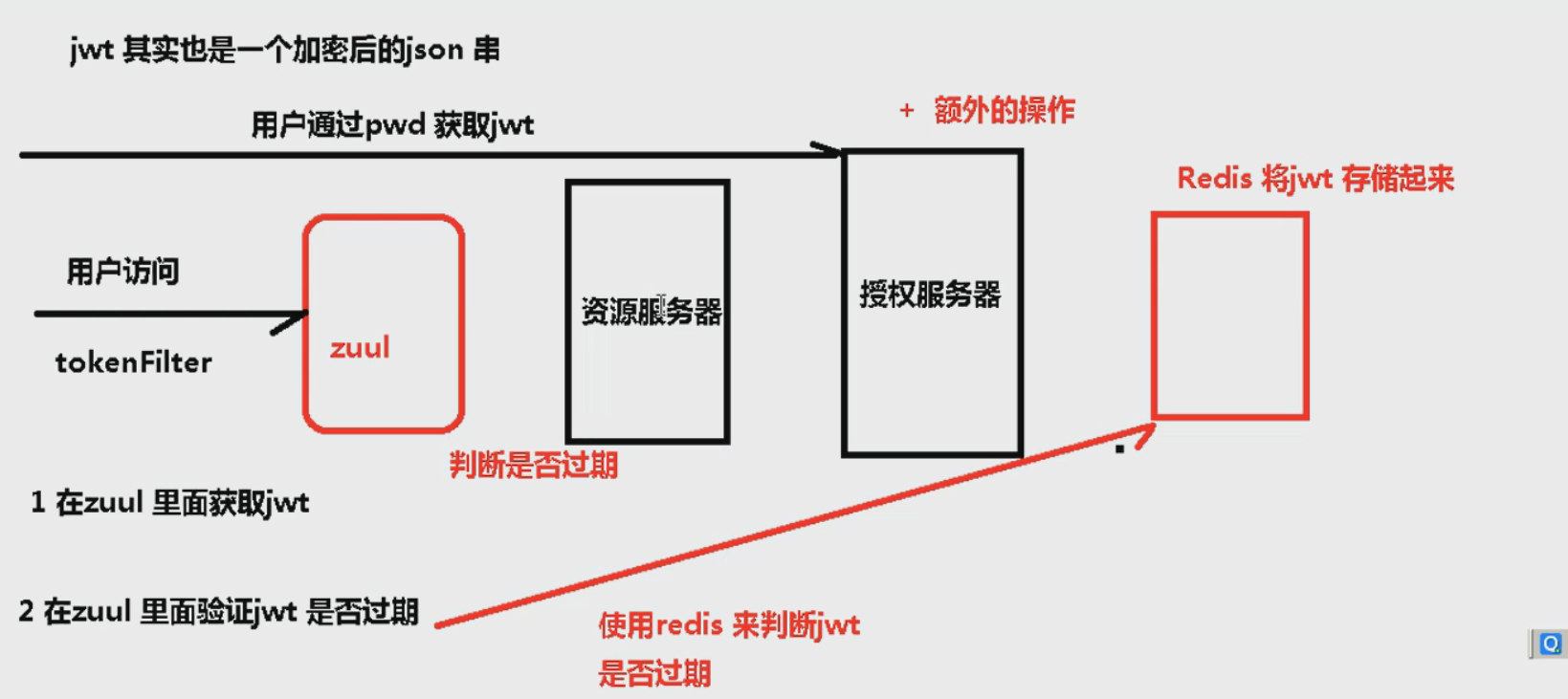




## 让zuul称为第三方

1、





# jwt（json web token）

## 跨域认证的问题

## JWT 的原理

JWT 的原理是，服务器认证以后，生成一个 JSON 对象，发回给用户，就像下面这样。

{

"姓名": "张三",

"角色": "管理员",

"到期时间": "2018年7月1日0点0分"

}

以后，用户与服务端通信的时候，都要发回这个 JSON 对象。服务器完全只靠这个对象认定用户身份。为了防止用户篡改数据，服务器在生成这个对象的时候，会加上签名（详见后文）。

服务器就不保存任何 session 数据了，也就是说，服务器变成无状态了，从而比较容易实现扩展。

## JWT 的数据结构

实际的 JWT 大概就像下面这样。



它是一个很长的字符串，中间用点（.）分隔成三个部分。注意，JWT 内部是没有换行的，这里只是为了便于展示，将它写成了几行。

JWT 的三个部分依次如下。

Header（头部）

Payload（负载）

Signature（签名）

写成一行:Header.Payload.Signature



### 下面依次介绍这三个部分。

Header 部分是一个 JSON 对象，描述 JWT 的元数据，通常是下面的样子。

{

"alg": "HS256",

"typ": "JWT"

}

面代码中，alg属性表示签名的算法（algorithm），默认是 HMAC SHA256（写成 HS256）；typ属性表示这个令牌（token）的类型（type），JWT 令牌统一写为JWT。

最后，将上面的 JSON 对象使用 Base64URL 算法（详见后文）转成字符串。

### Payload

Payload 部分也是一个 JSON 对象，用来存放实际需要传递的数据。JWT 规定了7个官方字段，供选用。

|  |
| --- |
| iss (issuer)：签发人  exp (expiration time)：过期时间  sub (subject)：主题  aud (audience)：受众  nbf (Not Before)：生效时间  iat (Issued At)：签发时间  jti (JWT ID)：编号 |

除了官方字段，你还可以在这个部分定义私有字段，下面就是一个例子。

|  |
| --- |
| {  "sub": "1234567890",  "name": "John Doe",  "admin": true  } |

注意，JWT 默认是不加密的，任何人都可以读到，所以不要把秘密信息放在这个部分。

这个 JSON 对象也要使用 Base64URL 算法转成字符串。

### Signature

Signature 部分是对前两部分的签名，防止数据篡改。首先，需要指定一个密钥（secret）。这个密钥只有服务器才知道，不能泄露给用户。然后，使用 Header 里面指定的签名算法（默认是 HMAC SHA256），按照下面的公式产生签名。

HMACSHA256(

base64UrlEncode(header) + "." +

base64UrlEncode(payload),

secret)

算出签名以后，把 Header、Payload、Signature 三个部分拼成一个字符串，每个部分之间用"点"（.）分隔，就可以返回给用户。

### Base64URL

前面提到，Header 和 Payload 串型化的算法是 Base64URL。这个算法跟 Base64 算法基本类似，但有一些小的不同。

JWT 作为一个令牌（token），有些场合可能会放到 URL（比如 api.example.com/?token=xxx）。Base64 有三个字符+、/和=，在 URL 里面有特殊含义，所以要被替换掉：=被省略、+替换成-，/替换成\_ 。这就是 Base64URL 算法。

## JWT 的使用方式

客户端收到服务器返回的 JWT，可以储存在 Cookie 里面，也可以储存在 localStorage。

此后，客户端每次与服务器通信，都要带上这个 JWT。你可以把它放在 Cookie 里面自动发送，但是这样不能跨域，所以更好的做法是放在 HTTP 请求的头信息Authorization字段里面。

Authorization: Bearer <token>

另一种做法是，跨域的时候，JWT 就放在 POST 请求的数据体里面。

j

# 附录

## springboot项目的打包（jar—>docker镜像）

### springboot项目的配置文件中变量的来源

1、控制台（优先级最高）

2、系统环境变量（次之）

例如：在yam文件中写user.name，将会读到计算的名称。

3、来自配置文件（最低）

例如：java -jar eureka-server-1.0.jar --server.port=8080 //这里指定配置文件中启动的端口。

注：在docker中可以使用-e来使用环境变量。

例如：-e SERVER\_PORT=8080 //运行载docker中，是如何取到该值的呢？

例如：

|  |
| --- |
| server:  port: ${SERVER\_PORT} #该值以后来自环境变量，在docker启动时，通过-e参数指定 |

### 运行jar包

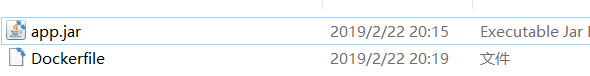
运行：java -jar jar包的名字

启动jar时， 附带端口：

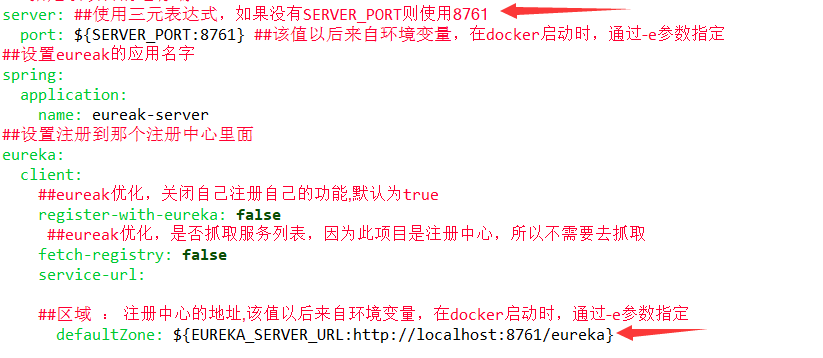
### 配置文件和jar包

1、创建文件夹eureka-service

两个文件



2、jar包的yml配置文件中：



3、Dockerfile中镜像

|  |
| --- |
| FROM openjdk:8-alpine  ENV WORKDIR /usr/src/myapp ##定义一个WORKDIR变量  COPY . ${WORKDIR} ##将当前文件夹下的内容复制到/etc/user/app中  WORKDIR ${WORKDIR} ##切换到当前的目录  CMD [ "java","-jar","app.jar" ] ##执行app.jar |

4将eureka-service移动到/usr/local中

执行构建命令：docker build ./eureka-service/ -t "eureka:1.0"

5运行镜像

docker run –name eureka -p 8761:8761 -d eureka：1.0

6、开启服务

